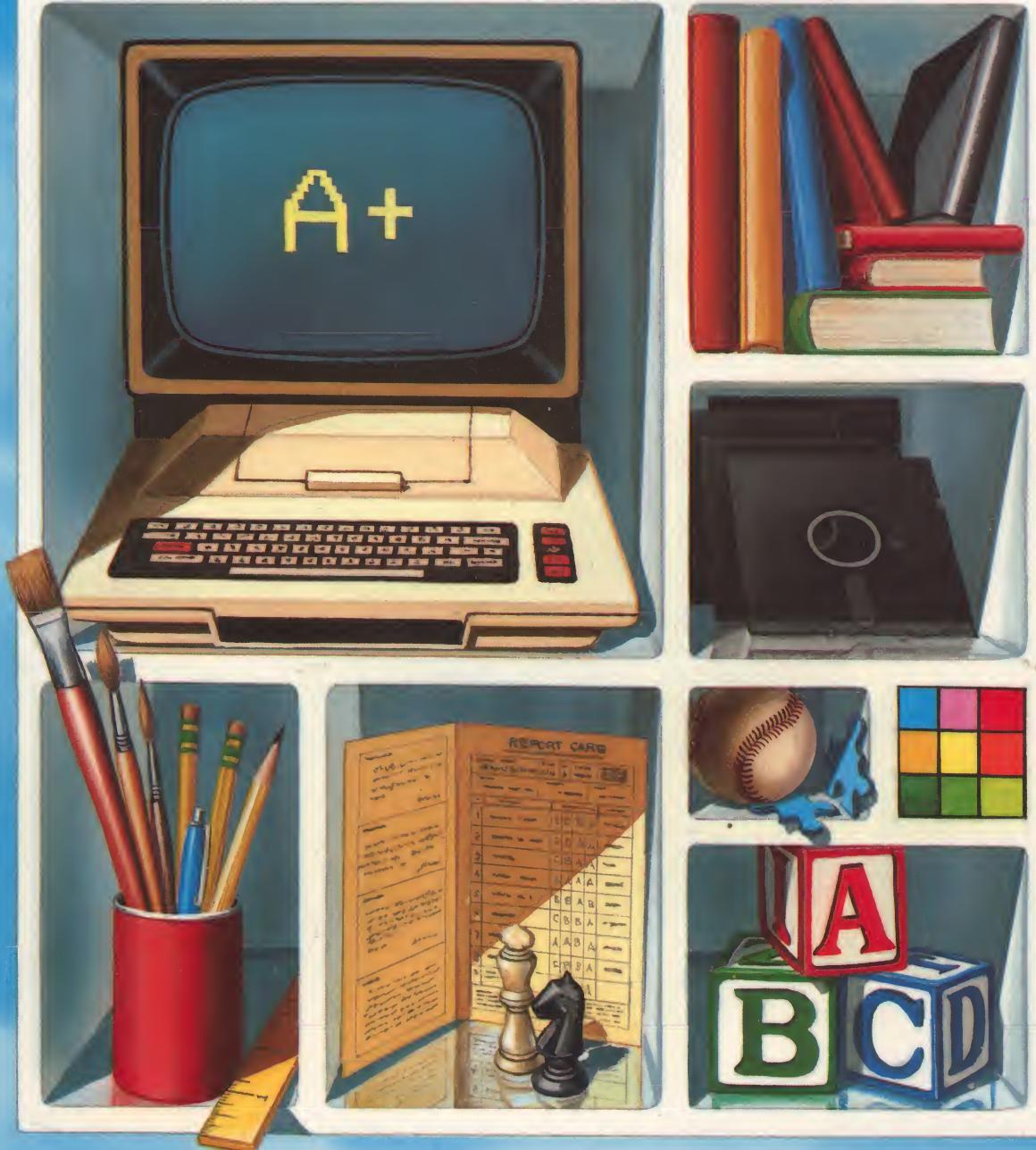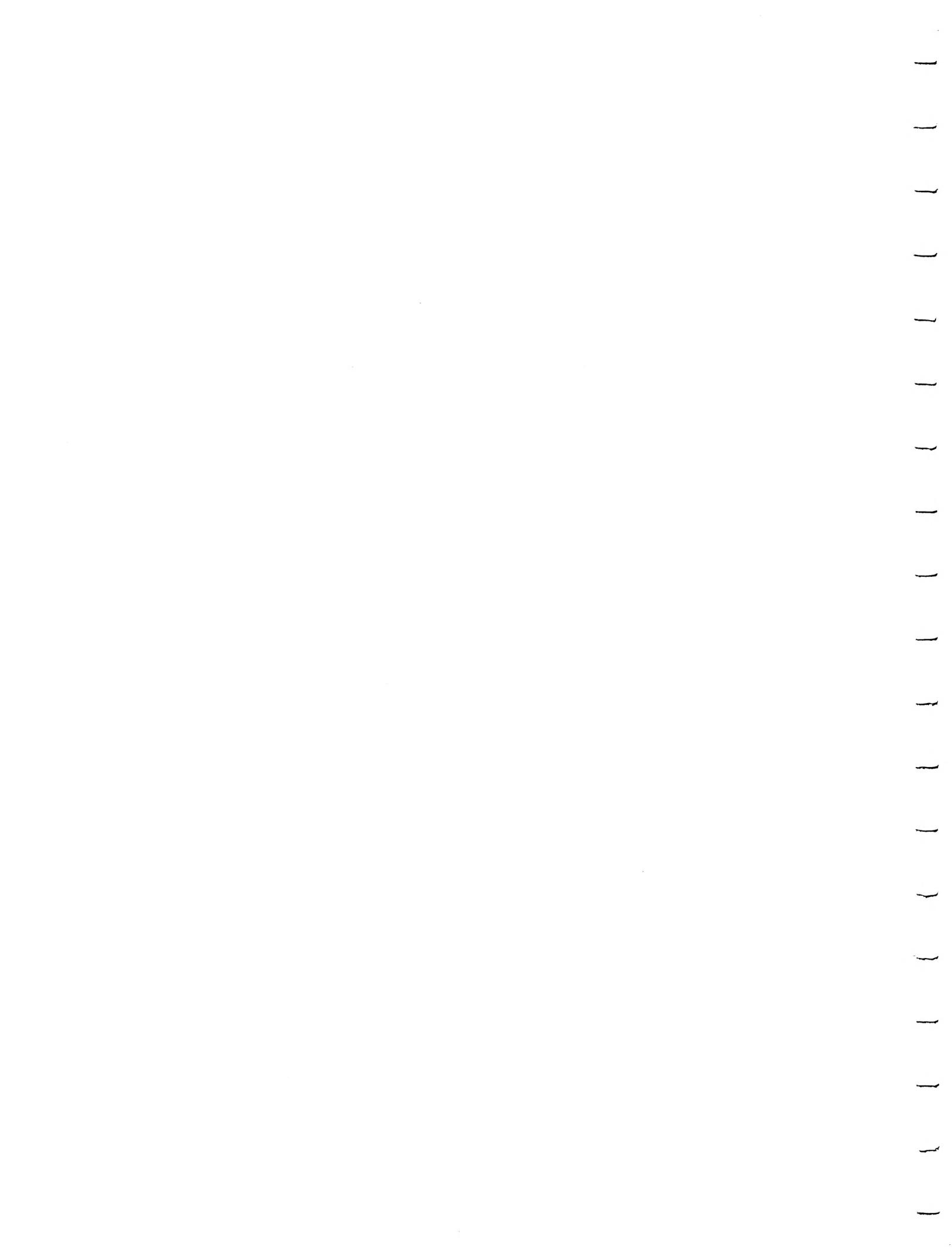John M. Reisinger

A+ Programming
in Atari BASIC

A+ Programming
in Atari® Basic

A+ Programming
in Atari® Basic

John M. Reisinger

Patch American High School
Stuttgart/Vaihingen
Federal Republic of Germany

ATARI® is a registered trademark of ATARI, Inc., a Warner Communi-
cations Company, Sunnyvale, California.

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Acknowledgments

Preface

To The Student:

The eighteen chapters of this book are intended as a beginning course in BASIC programming. Our intent throughout the book is to provide a variety of experiments to help you learn how to program in BASIC. If we suggest that you try something on the computer, then please do! Only by actually typing in programs and modifying them can you learn programming. We think that you'll find the whole experience exciting and rewarding. Have fun!

To The Teacher:

This book is not intended as an attempt to impose any particular programming philosophy on you or your students. You may use it in any way that meets their needs. Certainly you have your own favorite programs and exercises that you will want to include in your course. We have purposely deemphasized the more traditional business-related applications of BASIC in favor of more motivating graphics and sound programs.

We hope that you and your students will enjoy BASIC programming on the ATARI as much as we have!

Contents

Chapter 5 - The INPUT Statement

Chapter 6 - READ, DATA, and RESTORE STATEMENTS

Chapter 7 - Transfer of Control Statements

Chapter 8 - Anatomy of a Loop

Chapter 9 - FOR...NEXT Loops

Chapter 10 - Flowcharts

Chapter 11 - Writing Understandable Programs

Chapter 12 - More about Graphics

Chapter 13 - Numeric Functions

Chapter 14 - String Functions

Chapter 15 - Variables with One Subscript (Arrays)

Chapter 16 - Variables with Two Subscripts (Matrices)

Chapter 17 - Subroutines

Chapter 18 - Review

Operating the Computer

Welcome to the world of ATARI! We feel certain that learning to program an ATARI personal computer is going to be an exciting and rewarding experience for you. Please think of this book as a guide to your learning; you're going to learn to program not so much by reading these pages but by *using the computer*.

But what is a *program*? A program is a sequence of instructions to the computer that tells it what steps to perform in order to produce a desired result, or *output*. The instructions are written in some kind of *computer language*; the language we are learning is called BASIC (Beginners' All-purpose Symbolic Instruction Code). A computer program is an example of *software*; the computer itself, printer, TV screen or monitor, and disk drive are examples of *hardware*.

We just referred to the results of a program as output. A computer or computer system has four main functions:

1. Input: This is the process of "feeding" a program or data for the program into the computer system. On the ATARI, the *keyboard* is the main input device. The cassette recorder and disk drive may also be used as input devices.

2. Storage: A program that is typed into the computer is stored in the computer *memory* until we turn the computer off. We may also store the program permanently on a cassette tape or on a diskette.

3. Processing: The Central Processing Unit (CPU) of a computer is what makes it all work. The CPU consists of a

control unit, which coordinates all the computer activity, and an *arithmetic-logic unit,* which does the calculations and makes decisions.

4. Output: This refers to the results of a program. Output may be a screen display, a printed sheet, or a set of data on a tape or diskette, as well as other forms and means.

Turning on the Computer

We're going to assume now that you have access to the *ATARI Operators Manual* for your particular computer in case you have problems, but our comments should be enough to get you started. Follow these steps:

1. **Plug the AC power adapter into a 110 volt outlet. Plug the small end of the adapter into the jack labeled POWER IN on the panel on the side (or back, depending on the model) of the computer console.**

2. **If you are using a normal TV set for the screen display, then plug the TV cord that is permanently attached to the back of the computer console into the jack labeled COMPUTER on the TV switch box.**

   If you are using a *monitor* for the screen display, then plug one end of the monitor cable into the jack labeled MONITOR on the computer, and plug the other two ends of the cable into the audio and video jacks on the monitor.

3. **(400, 800 and 1200XL models only.) Insert the BASIC cartridge in the appropriate cartridge slot of the computer.**

4. **Turn on the TV set (or monitor), and turn on the POWER ON/OFF switch on the computer console. The power indicator light on the keyboard should light up red.**

If you followed the steps correctly, then the display in Figure 1-1 should appear on the screen. Look at the bright square directly below the "R" in "READY" on the screen.

This square is called the *cursor* and indicates our position on the screen. Type something, and you will see that the "typing" starts at the cursor and that the cursor moves with what you are typing, always showing the next available space. Now we need to become familiar with the keyboard.

The Keyboard

You can see that the ATARI keyboard is very much like a normal typewriter keyboard but with a few extra keys added. Let's concentrate first on the keys that type the letters of the alphabet, the numerals, and the punctuation marks. We'll call these keys the

```
READY
■
```
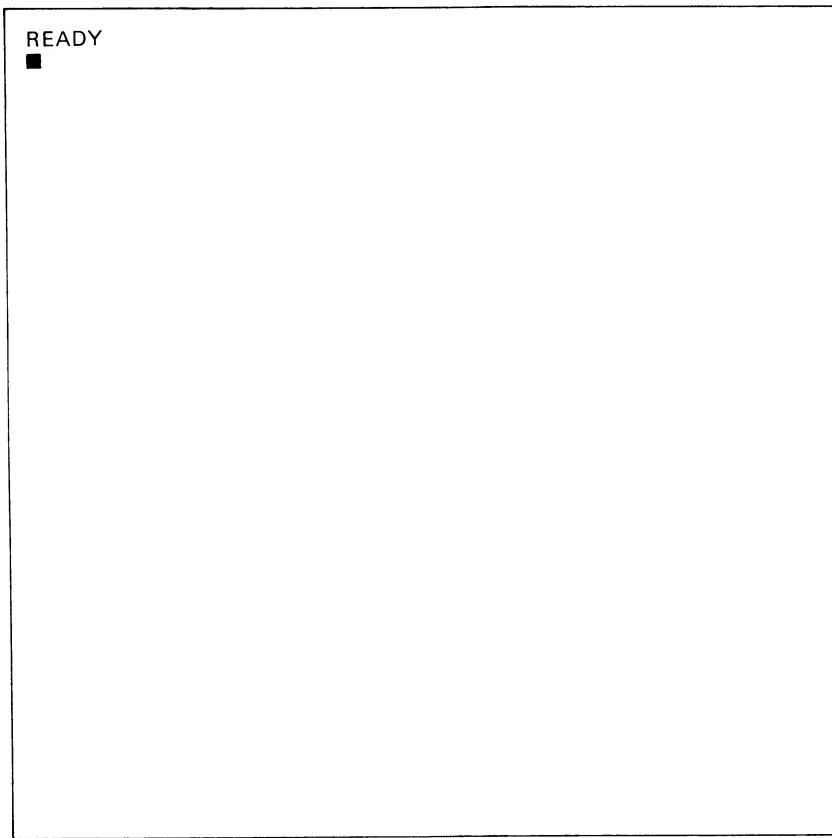
FIGURE 1-1   The ATARI BASIC Display Screen

character keys because each one will print one or more charac-
ters on the screen. Here is a chart of just the character keys,
without the special keys at the edges of the keyboard. The long
bar at the bottom is the space bar. Compare the chart with the
actual keyboard.

| ! 1 | ″ 2 | # 3 | $ 4 | % 5 | & 6 | ′ 7 | @ 8 | (9 |) 0 | < | > |

| Q | W | E | R | T | Y | U | I | O | P | ↑ – – | ↓ \| = |

| A | S | D | F | G | H | J | K | L | : ; | ← \ + | → ^ * |

| Z | X | C | V | B | N | M | [, |] . | ? / |
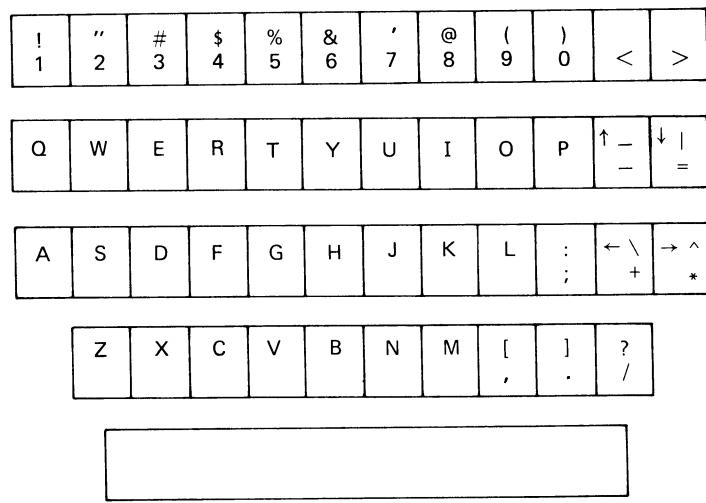
FIGURE 1-2   The ATARI Character Keyboard

Now type whatever you want. See if you can completely fill up the screen. Type, type, type! (Two fingers count here.) If you don't already know the keyboard, you'll be surprised at how fast you learn it. Note that the cursor automatically returns to the beginning of the next line when you reach the right edge. Have fun!

Could you print the characters on the upper half of the keys, such as $, &, ?, etc.? If you couldn't, try them now. Simply hold down either SHIFT key (lower left and right corners of the keyboard) while you press the desired key. This works exactly like a normal typewriter. How about lowercase (small) letters? Press the CAPS (CAPS/LOWR on some models) key (right end of third row from top) to change to lowercase. Now the SHIFT key will switch you back to uppercase. Press SHIFT and CAPS (CAPS/LOWR) together to switch back to uppercase permanently. Practice these things now.

In addition to the character keys, the ATARI has several *special function keys*: keys that don't print a character by themselves but control certain *screen editing* functions. Screen editing is the process of modifying what is displayed on the screen. Figure 1-3a and Figure 1-3b show the special function keys for
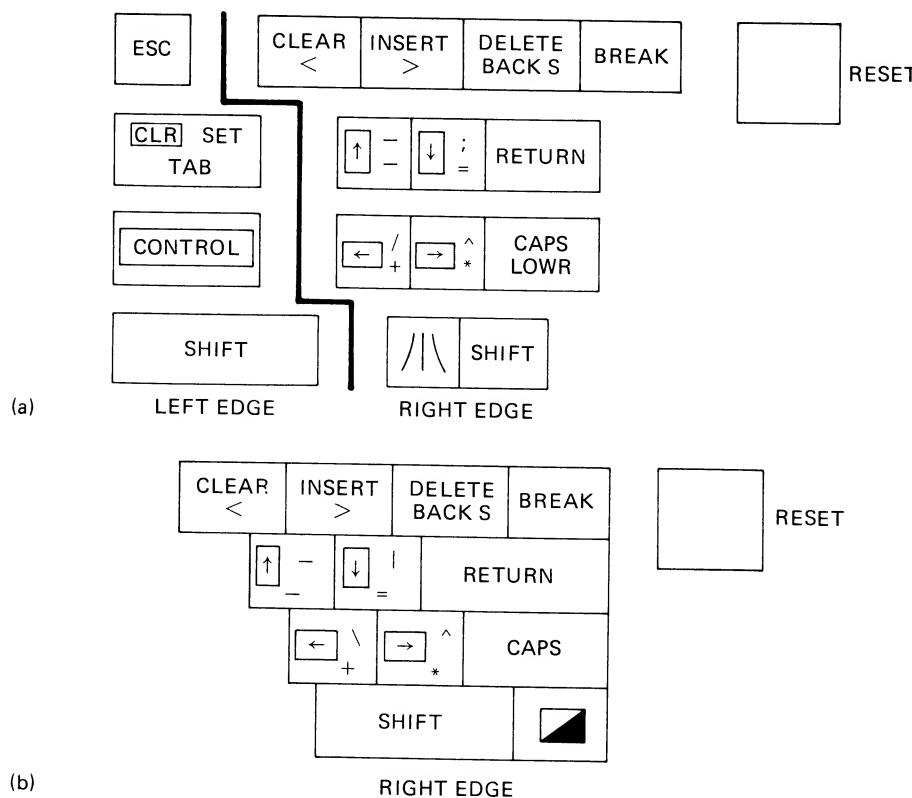
FIGURE 1-3   The ATARI Special Function Keys

the older and newer ATARI models. Notice that we have split the keyboard to show only these special function keys. They are to the left and to the right of the character keys. Take some time now to be sure that you can find the keys shown in the chart on the actual keyboard.

Note particularly the special function keys: CLR , CONTROL , ↑ , ↓ , ← , and → . These are extremely important in screen editing.

Now let's see what the special function keys can do. Type a few lines on the screen, and then practice using each special function as it is discussed. Don't press the RETURN key during these exercises.

CLEAR SCREEN: Press either SHIFT and CLEAR, or CONTROL and CLEAR at the same time. The screen goes blank, and the cursor returns to "home," that is, the upper left corner of the screen.

MOVE CURSOR: Hold down the CONTROL key and press the arrow key with the direction you want to go. By the way, have you discovered that if you hold a character key down for more than a second, it keeps printing its character? The same thing is true about moving the cursor. Now move the cursor back over a word on the screen and change some or all of the letters. See how easy it is to make corrections? Get used to this superb feature of the ATARI—you'll need it!

BACKSPACE: Press the DELETE/BACKSPACE key by itself. The cursor moves from right to left, *erasing as it goes.* Be careful! If you don't want to erase, use the move cursor function.

FORWARD SPACE: Press the *space bar.* The cursor moves from left to right, *erasing as it goes.* The preceding word of caution applies here, too.

INSERT A LINE: Move the cursor to the beginning of the line *below* where you would like the line inserted. Press SHIFT and INSERT at the same time. The line where you place the cursor moves down one line, and a blank line is inserted *above* it.

DELETE A LINE: Move the cursor to the beginning of the line you want to delete. Press SHIFT and DELETE/BACK S (delete/backspace) at the same time. Poof! The entire line vanishes. Try clearing the screen this way by holding both keys down for more than a second.

INSERT A CHARACTER: Move the cursor to the character (or space) *immediately to the right* of where you want the space. Press CTRL and INSERT at the same time. The character under the cursor and all of the characters to its right move *one space to the right,* and the cursor appears in the empty space. Remember that the space appears *to the left* of where you place the cursor.

DELETE A CHARACTER: Move the cursor to the character you want to delete and press CTRL and DELETE/BACK S simultaneously. The character under the cursor disappears, and all of the characters to its right move *one space to the left.*

TABULATE: Press the TAB key, and the cursor will move to the next TAB setting, just as on a normal typewriter. To set a tab, move the cursor to the desired location and press SHIFT and TAB simultaneously. To clear a tab, move the cursor to the location to be cleared and press CTRL and TAB. Tabs that you set in this manner will be cleared when the computer is turned off.

INVERSE VIDEO: The inverse video key is labeled ◪ or 𝄢 . It is located either next to the right-hand SHIFT key or above the keyboard. Press the key. Now everything that you type is in inverse video; that is, the colors of print and background are reversed. To get out of inverse video, press the inverse video key again.

Don't be frustrated by this laundry list of special functions— you'll learn to use them quickly as you begin to type programs into the computer. Do refer to the list frequently, though, because using the excellent screen editing features of the ATARI will save you much time.

Control Graphics

There are other characters that can be printed on the screen that are not indicated on the keyboard: the *control graphics charac- ters.* These are printed by holding down the CONTROL key and any of the alphabetic keys (A,B,C...,X,Y,Z) or the comma (,), period (.), or semicolon (;). Figure 1-4 is a chart of the control graphics characters.

The control graphics characters can be used to create a great variety of graphics designs—experiment with them now before we continue.
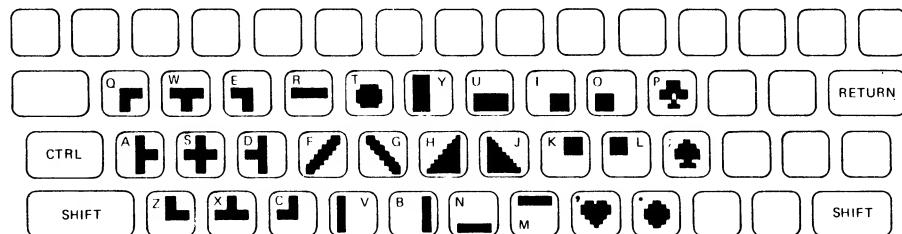
FIGURE 1-4 The ATARI Control Graphics Keyboard

Yet another set of characters, the *control characters*, can be printed by using the ESC key in conjunction with one or two other keys. This allows you to actually print the cursor control arrows and the CLEAR and INSERT arrows on the screen. Using these characters is beyond the scope of a first course in ATARI BASIC, but you can refer to the *ATARI Basic Reference Manual* for instructions on printing them.

Now let's use some BASIC commands. A quick way to clear the screen is to press the (SYSTEM) RESET key on the right side of the keyboard. You should get the display from Figure 1-5: "READY" in the upper left corner of the screen, with the cursor under the "R."

Basic
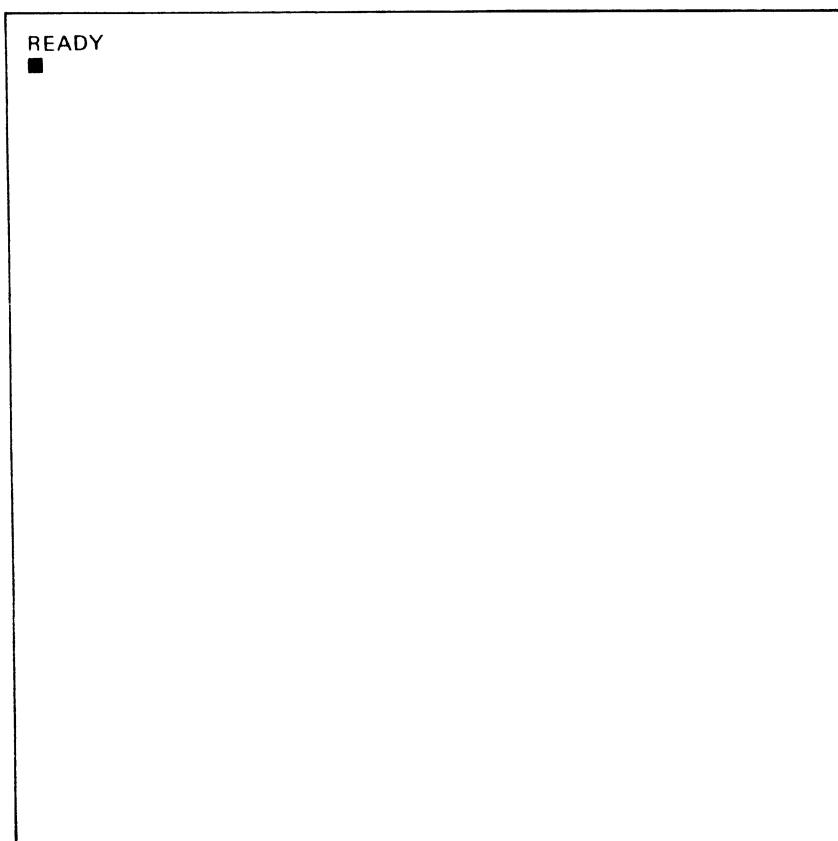
Figure 1-5 The ATARI BASIC Display Screen

We are in DIRECT, or immediate, mode. In this mode, we can give individual BASIC language commands and have them EXECUTED (performed) immediately by pressing the RETURN key. Suppose you type

```
PRINT 13+17
```

and press the RETURN key. The computer's response will be:

```
30
```

```
READY
■
```

You're using the computer as a desk-top calculator! You can practice some calculations now, but first you should know that the symbol for multiplication is "*", and the symbol for division is "/". For example, we write "2 times 3" as "2*3", and we write "10 divided by 5" as "10/5". If the computer responds with "ERROR— PRINT...", just ignore it at this point and try again. You can clear the screen by pressing SHIFT and CLEAR at the same time. Notice that the RETURN key is performing a different function in direct mode: it *activates* the computer. The computer's response, of course, depends on what command(s) we have given it.

If you have a printer attached to your computer, you can print the results of your calculations on the printer by typing LPRINT instead of PRINT. If you would like the problem printed, too, enclose the problem in quotation marks. In ATARI BASIC, anything that is placed in quotes after the word PRINT or LPRINT will be printed exactly as it appears in the quotes. Think of quotes as performing a duplicating function. Here is an example. You type

```
LPRINT "2 + 3 is",2+3
```

and press RETURN, and the following is printed on the printer:

```
2 + 3 is   5
```

```
READY
■
```

(If you typed PRINT instead of LPRINT, the same thing will be printed on the screen.)

Chapter 2 is all about printing—this is just a brief introduction. Have some fun now seeing what calculations you can print on the screen and/or printer.

First, be sure your ATARI 410 Program Recorder is correctly attached to the computer:

1. **Plug the recorder DATA CORD into the jack labeled PERIPH-ERAL on the side panel of the ATARI console.**

2. **Plug the recorder power cord into an ordinary 110V power source.**

Now, follow these steps:

1. **Insert the cassette tape with the program you want to load into the Program Recorder with the recording surface toward you and the label *for the side of the tape that you want* right side up so that you can read it.**

2. **Push REWIND on the recorder and wait until the tape stops.**

3. **Push the tape counter reset button until it reads 000, then push STOP EJECT once.**

4. **On the computer keyboard type CLOAD and then press the RETURN key. You will hear *one* beep.**

5. **Push PLAY on the Program Recorder. Now push the RETURN key on the computer keyboard again. When the tape counter reaches about 8, you will begin to hear a series of beeps. This means the program is loading. When it is finished, the word READY will appear on the screen.**

6. **Push STOP on the Program Recorder and remove the tape.**

7. **You may *"run"* the program by typing RUN and then pressing RETURN.**

8. **It sometimes happens that a tape does not load properly. In this case, repeat steps 1–7 above. Sometimes it seems to help to type LPRINT and press the RETURN key at the beginning of the sequence.**

In order to *load* a program from a diskette, you must have the ATARI Disk Drive attached to your computer (with at least 16K RAM), and you must also load DOS (Disk Operating System). To attach the disk drive to the computer proceed as follows:

1. **Plug one end of a data cord into the jack labeled PERIPHERAL on the ATARI console. Plug the other end into *either* of the jacks labeled I/O CONNECTORS on the back of the disk drive unit. *Note:* If you also want to use the recorder, you may plug its data cord into the other I/O CONNECTOR jack.**

2. **Plug the small plug on the AC power adaptor into the jack**

labeled PWR. on the back of the disk drive unit, and the other end into an ordinary 110V wall socket.

To load DOS (a program that allows the computer to communicate with the disk drive), do the following:

1. **Press the PWR. ON/OFF switch on the front of the disk drive. The two red lights will glow, the drive will whir for about seven seconds, and then the *upper* red light will go out. The *lower* light will still glow.**

2. **Open the door in the front of the disk drive.**

3. **Insert a diskette that contains DOS (such a disk is shipped with the computer) with the label facing up and the notch on the left. Be careful not to touch the recording surface of the disk through the openings in the black protective jacket. Close the door of the disk drive.**

4. **Turn the computer console power switch on. The disk drive will whir for a few seconds, then the word READY will appear on the screen. You are now ready to load a program.**

5. **Remove the DOS diskette from the disk drive and return it to its protective paper sleeve.**

Let's suppose that the name of the program you want to load is BLASTOFF. Follow these steps to load it:

1. **Press the PWR. ON/OFF switch on the disk drive.**

2. **When the *upper* red light goes out, open the door of the disk drive, insert your diskette (label up, notch to the left), and then close the door of the disk drive.**

3. **Type: LOAD "D:BLASTOFF" and press the RETURN key. The disk drive will whir for a few seconds, then the word READY will appear on the screen. Your program has been loaded. Remove the diskette from the disk drive, *replace it in its protective paper sleeve,* and turn the disk drive off.**

4. **You may *run* the program by typing the word RUN and then pressing the RETURN key.**

After you have *run* the program(s) that you have loaded, type the word LIST and press the RETURN key. The list of instructions that produced the display will go by on the screen. To stop the list, hold down the CONTROL key while you press the "1" key. Do the same to start the list again.

Learning to write such a list of instructions to the computer (a program) is what this book is all about. You've learned a lot already; now you're ready to start programming!

EXERCISES

1. The four main functions of a computer system are:

   _____, _____,

   _____, and _____.

2. A disk drive is an example of ( ) hardware, ( ) software.

3. A program that draws a picture of a disk drive is an example of
   ( ) hardware, ( ) software.

4. To clear the screen, we press the SHIFT key and the _____
   key.

5. If we move the cursor to a character in a line on the screen and then
   press the CTRL key and the INSERT key at the same time, then a
   space appears to the ( ) left ( ) right of the character that was
   under the cursor.

6. [( ) True ( ) False] Pressing the RETURN key will move the
   cursor to the beginning of the line you were on.

7. Which of the following should be typed to load a program called
   TESTING from a diskette?
   ( ) "D:TESTING"
   ( ) LOAD "D:TESTING"
   ( ) LOAD TESTING

8. The symbol for multiplication is _____.

9. The CPU of a computer has two parts: the _____ unit and the

   _____ unit.

10. [( ) True ( ) False] We could load a program called BLASTOFF
    from a cassette tape by typing CLOAD "BLASTOFF".

11. [(   ) True   (   ) False]   Inverse video means that characters will be printed upside down.

12. You have just typed PRIMT "3 + 7 is",3+7 and you realize that you have made a mistake. You should
(   ) Press SHIFT and CLEAR and start over.
(   ) Move the cursor (press CTRL and the left arrow) to the M, type N, and then press RETURN.

13. A sequence of instructions to the computer to have it produce a

certain result is called a ＿＿＿＿＿＿ .

Printing

In this chapter we are going to cover how to print the results of computer computations on the screen and on the printer. Such results are known as *output*.

Because we will be using the BASIC language, if you are using an ATARI 400, 800, or 1200 XL, check now to be sure that the BASIC cartridge is inserted in the cartridge slot of the computer console.

Type the following *program* on the computer keyboard exactly as it appears below. At the end of each line, press the RETURN key and the *cursor* will move to the beginning of the next line. When you have finished, type the word RUN and press the RETURN key. Please be careful to use upper and lowercase letters exactly as they appear below.

```
100 PRINT "ADD TWO NUMBERS"
110 PRINT
120 PRINT "The sum of 5.3 and 9.8 is"
130 PRINT 5.3+9.8
```

Your *run* of the program should have looked like this on the screen:

```
RUN
ADD TWO NUMBERS

The sum of 5.3 and 9.8 is
15.1

READY
■
```

13

(READY is always the computer's signal to us that it has finished a task and is ready for a new one.)

Now clear the screen by holding down either SHIFT key while you press the CLEAR/< key. Our program is stored in the computer, and we can see it again by typing LIST and pressing the RETURN key. Do this now.

Let's change the problem the computer did for us by changing lines 120 and 130 of the program to look like this:

```
120 PRINT "The sum of 23.98 and 117.67 is"
130 PRINT 23.98+117.67
```

We don't need to retype the entire line—we can just move the cursor to the numbers and change them. Remember, to move the cursor, hold down the CTRL key and press one of the arrow keys. The cursor will move in the direction of the arrow. To insert a space (this is necessary because our new numbers have more digits than the old ones), move the cursor to the *right* of where you need the space, then hold down the CTRL key and press the INSERT/> key once for each space that you need. Be sure to press the RETURN key at the end of each line. Notice that line 120 automatically extends to the next line.

After you have made the changes, type the word LIST and inspect the new program to see if you have made any mistakes. If not, type RUN to see your results. The output should look like this:

```
RUN
ADD TWO NUMBERS

The sum of 23.98 and 117.67 is
141.65

READY
■
```

Now change the program to look like the following:

```
100 PRINT "MULTIPLY TWO NUMBERS"
110 PRINT
120 PRINT "The product of 5 and 9.3 is"
130 PRINT 5*9.3
```

RUN the program. The output should look like this:

```
RUN
MULTIPLY TWO NUMBERS

The product of 5 and 9.3 is
46.5

READY
■
```

As you type in these programs, you are using the computer in the *deferred mode*; that is, the instructions are not executed immediately, as in the *direct mode*. As soon as we type RUN, the computer is in the *execute* (or run) *mode*. We have just four modes: memo pad (400 and 800 models only), direct, deferred, and execute.

You may have experienced in the previous exercises that the computer printed something like the following on the screen:

```
100 PRINT MULTIPLY TWO NUMBERS"
100 ERROR-   PRINT MULTIPLY TWO NUMBERS"
```

In this case the cursor will appear superimposed on the T in TWO. The computer is telling us that it cannot completely understand line 100. We forgot the quotation marks! To correct this error, follow the steps below. This will be the standard procedure for correcting error messages in this text.

TABLE 2-1   Error Correction Procedures

1. **Move the cursor to the beginning of the original line that contains the error. (Use the CONTROL key and the arrow keys.)**
2. **Correct the line using cursor control. Press the RETURN key after all mistakes in the line are corrected.**
3. **Press the RETURN key to move the cursor to the beginning of the line with the error message in it. Hold down either SHIFT key and press the DELETE/BACKSPACE key. The line will disappear. The error should be corrected.**
4. **Type LIST to check that the program has been corrected. If not, repeat the steps above.**

Suppose that we had typed the following message:

```
100 PRIMT "MY NAME IS ATARI"
```

(Notice the spelling error.) The computer will type the following message:

```
100 ERROR-    PRIMT "MY NAME IS ATARI"
```

In this case, the cursor will appear superimposed on the first set of quotation marks. (The computer thinks that the word PRIMT is a *variable*, but more about that later.) So we correct our statement by using the steps outlined in Table 2-1. Whenever we type a command that the computer cannot understand, we will get an error message, and we should correct it using the method outlined.

QUESTION: Why won't we get an error message if we type

```
100 PRINT "MY NSME ID ATARI"
```

Do you have an idea?

Right! The ATARI prints whatever is inside quotation marks exactly as it appears, even if we make a typing error. It must, however, be able to understand the *reserved words* or *keywords* of ATARI BASIC. Learning to program in BASIC is largely a question of learning to put these keywords together in sensible statements. So far, we know three keywords: PRINT, RUN, and LIST.

As we begin to write programs, we will discover that we make frequent errors of another sort: The computer can understand our instruction, but it can't carry it out. In such a case, we will get an error message that tells us two things: (1) A *code number* for the type of error, and (2) the *line number* of the line in the program where the error occurred. For example, type in the following short program and run it.

To erase the previous program from the computer memory, type the word NEW and press the RETURN key.

```
100 PRINT "ERROR MESSAGE"
110 PRINT
120 GOTO 150
130 PRINT "THIS WON'T BE PRINTED"
```

Your RUN of the program should have looked like this on the screen.

```
RUN
ERROR MESSAGE


ERROR-     12 AT LINE 120
■
```

Now we need to know what error 12 is, so we consult the appendix in the back of this book or the *ATARI BASIC Reference Manual.* Error 12 means "line not found." Because this error occurred at line 120, we need to look at line 120 of our program. If it is not still on the screen, we can get it back by typing

```
LIST 120
```

and pressing the RETURN key. Line 120 is

```
120 GOTO 150
```

Because there is no line numbered 150 in the program, the computer certainly cannot go there! Thus, the error message. We can correct the program by adding the line

```
150 PRINT "I FIXED IT!"
```

Try this now.

As we learn more and more BASIC commands, we will make many different errors (to err is human!), and eventually we will have memorized the number codes for the more frequent ones. Think of the error messages as the computer's way of helping you and don't be frustrated when they occur. We all make many mistakes, but we can also correct them!

Line Numbers

You have noticed by now that all of the programs we have considered have one feature in common: Each line of the program starts with a number. Yes, these are the line numbers we referred to above. In general, the computer performs (executes) the commands given in the program statements in the same order as the numbers of the statements. We may use any numbers from 0 to 32767 (one less than 2 raised to the 15th power) in ATARI BASIC. Usually we number by 10s (or even 20s or 100s) so that we can insert additional lines later if needed. In this book, we will not use line numbers below 100. Try to guess

what the following program will produce, then type it in exactly
as it appears and run it to see if you are right. Then LIST the
program.

```
140 PRINT "ORDER"
100 PRINT "THIS"
120 PRINT "OUT"
130 PRINT "OF"
110 PRINT "IS"
```

Were you surprised? Notice that when we LIST a program,
the computer will automatically put the lines in numerical order.

Arithmetic Operators

In BASIC, the following symbols are used for arithmetic
operations:

- Addition   +
- Multiplication   *
- Subtraction   –
- Division   /
- Exponentiation   ^

Exponentiation is the process of raising a number to a pow-
er. We write 2 to the third power (2 cubed) as $2 \wedge 3$.

The symbols listed above are called *arithmetic operators*, or
simply *operators*. These five operators form the basis for all
mathematic computation on the computer.

We can practice using these symbols by typing in the follow-
ing instructions. After each line, press the RETURN key, and
the result of the calculation will be printed immediately. We are
using the direct, or immediate, mode.

```
PRINT 2+3
PRINT 2*3
PRINT 12-7
PRINT 28/4
PRINT 2^7
PRINT 5*(6+3)
PRINT (37+8)/(7-2)
PRINT 3.14*5*5
PRINT 8+4/2
PRINT (8+4)/2
```

Did you predict correctly the results of the last two lines?
Obviously we must be careful to put the operators in the correct
order.

The computer does operations in exactly the same order that we do them in algebra:

1. **Expressions in parentheses, beginning with the innermost parentheses and working outward**

2. **Exponentiation**

3. **Multiplication and division, in order from left to right**

4. **Addition and subtraction, in order from left to right**

Examples:

1)   3+5*2 ∧ 3-4/2*3
becomes	3+5*8-4/2*3
then	3+ 40-2 *3
then	3+ 40- 6
then	43- 6
then	37

2)   (8/(3+1)+10)/4
becomes	(8/4+10)/4
then	(2 +10)/4
then	12 /4
then	3

3)   (3+5)*(9-6)+12/4+2
becomes	8*3+12/4+2
then	24+ 3 +2
then	27 +2
then	29

4)   2∧3∧4-1
becomes	8 ∧ 4-1
then	4095.99999-1
then	4094.99999

(*NOTE:* We often get a slight rounding error in exponentiation. The correct answer is 4095. We will discuss techniques for correcting this error in CHAPTER 13.)

5)   91+87/2
becomes	91+43.5
then	134.5

6)   (91+97)/2
becomes	178/2
then	89

If you have a printer attached to your computer, then you can have the results of your programs printed on the printer by making some very simple changes in the programs. Consider the

Printing on a Printer

program at the beginning of this chapter:

```
100 PRINT "ADD TWO NUMBERS"
110 PRINT
120 PRINT "The sum of 5.3 and 9.8 is"
130 PRINT 5.3+9.8
```

To have the results printed on the printer, simply replace each PRINT in the program with LPRINT. (This is an abbreviation for LINE PRINT because printers are often referred to as *line printers*.) Your program will now look like this:

```
100 LPRINT "ADD TWO NUMBERS"
110 LPRINT
120 LPRINT "The sum of 5.3 and 9.8 is"
130 LPRINT 5.3+9.8
```

Run the program and watch the results be printed on your paper. The sheet of printed results is known as a *hard copy* of your results. If you would like a hard copy of your program statements, simply type

```
LIST "P:"
```

and press RETURN. "P:" is simply an abbreviation for the printer. Don't forget the colon! The ATARI will automatically supply quotation marks at *the right end of a statement* if we leave them off. For example, we could just as well type

```
LIST "P:
```

and we would get the list, or hard copy, of our program on the printer.

Print Zones

The ATARI provides several ways to space text on the screen or the printed page. One of the ways is *zone spacing*. To get an idea of how it works, type in the following program and then run it.

```
100 PRINT "BASE","HEIGHT","AREA"
110 PRINT "----","------","----"
120 PRINT 5,4,(5*4)/2
130 PRINT 9.2,6,(9.2*6)/2
```

Your results should look like this:

BASE	HEIGHT	AREA
5	4	10
9.2	6	27.6

The ATARI screen is divided into 4 *zones* vertically. The first three zones are 10 spaces wide, and the fourth zone is 8 spaces wide. Thus, the screen is normally 38 spaces wide. (There is a margin of 2 spaces on the left edge.) Commas (,) placed between the items in a print line tell the computer to use zone spacing.

An alternative type of spacing is *compact spacing*. In this case, we separate the items in a print line with semicolons (;) instead of commas. Type in and run the following program:

```
100 PRINT "2 X 3 = ";2*3
110 PRINT "6 X 5.7 = ";6*5.7
```

Your results should look like this:

```
2 X 3 = 6
6 X 5.7 = 34.2
```

Recall that whatever is enclosed in quotation marks in a print statement is printed *exactly* the way it appears in the quotation marks.

At this point you may want to convert the programs above to work on a printer, and you may also want to experiment with your own programs. Eighty-column printers that are compatible with the ATARI computers vary greatly in their capabilities, but most will have these characteristics:

1. **The printed characters are spaced ten per horizontal inch on the paper, and there are six lines per vertical inch, just as on a typewriter.**

2. **There are eight print zones of ten characters each across a line of the paper.**

3. **Compact spacing works exactly as it does on the screen; that is, no spaces are left between items in a PRINT statement that are separated by semicolons.**

4. **If a line in a program contains more than 38 characters (but fewer than 80), then it will of necessity overlap onto the next line on the screen, but it will be printed completely on one line of paper with the printer.**

5. Special styles of printing (condensed, elongated, underlined, italics, etc.) may be produced by adding appropriate commands to LPRINT statements.

You will need to refer to the operating manual for your particular printer and then experiment as much as possible.

EXERCISES

1. The command to list a program on a printer is _____.

2. Write the printout (output) for each of the following one-line programs. Try to compute the result *before* you use the computer.

    ```
    110 PRINT 5+7                    _____
    120 PRINT (5+7)                  _____
    130 PRINT (5+7)*3                _____
    140 PRINT 5+(7*3)                _____
    150 PRINT (6+(7*3))*2            _____
    160 PRINT (6+(7*3))*(2+5)        _____
    170 PRINT 2^4*(3+7)/4-5          _____
    180 PRINT 2^4*(3+7)/(4-5)        _____
    190 PRINT (5+1)^(7-4)*2/8-10     _____
    200 PRINT 2+2*2^2-2/2            _____
    ```

3. Correct each of the following BASIC statements. If a statement has no errors, write "correct."

    ```
    1,050 PRINT "TESTING"        _____
    55.1 PRINT 2*3               _____
    30 PRINT "SUM OF ANGLES      _____
    ```

```
727 PRINT "A = 3"          _____

33020 PRIMT "RESULT = ";1+1 _____
```

4. There are _____ print zones on the display screen.

5. [( ) True   ( ) False]   2 raised to the 5th power would be written 2POW5 in BASIC.

6. [( ) True   ( ) False]   The command 120 PRINT 2 x 3 = 6 will produce an error message.

7. Write the printout (output) for each of the following "one-line" programs. Try to anticipate the result *before* you use the computer.

```
110 PRINT "2 x 3 = 6"       _____

120 PRINT "2 x 3 = ";2*3    _____

130 PRINT "1234567";654321  _____
```

8. Write the printout (output) for each of the following "one-line" programs. Try to anticipate the result *before* you use the computer.

```
110 PRINT "DAY","MONTH"     _____

120 PRINT "DAY";"MONTH"     _____

130 PRINT 3*4;" + ";24/6    _____
```

9. Write the complete printout (output) of each of the following three programs. See if you can anticipate the printout without using the computer. Be sure to indicate the spacing into zones.

```
100 PRINT 1,2,3,4
110 PRINT 1;2;3;4

200 PRINT "BASE = ";5
210 PRINT "HEIGHT = ";7
220 PRINT
230 PRINT "AREA =";(5*7)/2

310 PRINT "FOR TRICKS"
300 PRINT "WATCH OUT"
```

10. As you type in a program, you are using the computer in the

_____ mode.

11. [( ) True ( ) False] One possible way to correct a line that produced an error message is to retype the entire line.

12. The symbols +, *, -, /, and ∧ are referred to as _____.

13. The largest line number that we may use in ATARI BASIC is

_____ .

14. The smallest line number that we may use in ATARI BASIC

is _____ .

15. Write a program that will produce the following printout. Be sure to have the computer do the calculations!

```
RUN

NAME        AGE
--------    ---
GEORGE       15
SANDRA       19
BILLY         8
--------    ---
  AVERAGE   14
```

3

Graphics and Sound

Two of the most attractive features of the ATARI computer are *color graphics* and *sound*. In this chapter we are going to learn how to use these features. By graphics we mean pictures or designs appearing on the screen. All the instructions for using color will be valid if you have a black and white TV set (or monitor) attached to your computer, but your graphics displays will be less attractive!

Graphics Modes

All graphics displays are produced by lighting up (in varying colors) dots, or *pixels*, on the screen. We have a choice of different sizes of dots to use, and these choices are referred to as *graphics modes*. To light up a dot on the screen, we must tell the computer the location of the dot. Type in and then run the following program and you will probably get an idea of the system that is used. We will explain the program in the following paragraphs. Line 100 is just for identification; it doesn't affect the program.

```
100 REM - Chapter 3, No. 1
110 GRAPHICS 3
120 COLOR 1
130 PLOT 0,0
140 DRAWTO 39,0
150 DRAWTO 39,19
```

```
160 DRAWTO 0,19
170 DRAWTO 0,0
180 COLOR 2
190 PLOT 1,1
200 DRAWTO 38,18
210 COLOR 3
220 PLOT 1,18
230 DRAWTO 38,1
```

Your run of the program should have produced an orange rectangle with a blue and green X in it. If it didn't, press the SYSTEM RESET key, then list the program and make corrections.

Now let's look at some of the instructions in the program. Consider the second line:

```
110 GRAPHICS 3
```

Change this line to

```
110 GRAPHICS 5
```

and then run the program again. What happens? The picture (graphics display) gets smaller. Or, to think of it in another way, we could get more dots on the screen. Let's make another change:

```
110 GRAPHICS 7
```

When we run the program this time, the picture gets smaller still. We have many more dots available, though, so that we could create a picture with much more detail.

We have just considered graphics modes 3, 5, and 7. Table 3-1 summarizes the number of dots available in each mode:

TABLE 3-1 Screen Formats - GRAPHICS 3, 5, 7		
Graphics Mode Number	*Columns (Left to Right)*	*Rows (Top to Bottom)*
3	40	20
5	80	40
7	160	80

The numbers of the rows and columns always start at zero (0), so that in graphics mode 3, for example, the columns are numbered from 0 to 39 and the rows are numbered from 0 to 19.

The location of a dot on the screen is always given by two numbers. The first number refers to the column (left to right), and the second number refers to the row (top to bottom). We always start numbering from the upper left corner of the screen.

There are just two commands to light up points on the screen on the ATARI. Look at line 130 of our program:

```
130 PLOT 0,0
```

This command lights up the pixel (dot) at the upper left corner of the screen. If we had wanted a pixel 10 columns from the left and 5 rows down lighted up, the command would have been

```
130 PLOT 10,5
```

Now look at line 140 of our program:

```
140 DRAWTO 39,0
```

This command lights up all of the pixels on the straight line between the pixel **0,0** and the pixel **39,0**. (Masochists could get the same effect by writing 40 separate PLOT statements!) Line 150 lights up all the pixels on the straight line between the pixel **39,0** and the pixel **39,19**. Similarly, lines 160 and 170 draw lines across the bottom of the screen and up the left side.

Now add the following line to your program:

```
175 STOP
```

This will do just what you think: The computer will stop executing the instructions at this point. Run the program. You should get just the orange rectangle. Do you see how the PLOT and DRAWTO statements work? If you would like to experiment a bit, try putting different numbers after the PLOT and DRAWTO statements in lines 130 to 170. See if your experiment works. You might get the following error message:

```
ERROR-    141 AT LINE 160
```

This will mean that you asked the computer to plot a pixel that was "off the screen"; that is, either the first or second number for the pixel was too big. Remember, the largest number for a column or row is *one less than* the number given in Table 3-1. (This is because the numbering starts at 0.)

If you are satisfied with your experiment(s), do the following: Remove line 175 from the program by typing

```
175
```

and then pressing the RETURN key. Line 175 will be gone from the program. You can check by LISTing the program. Now run the program again. Do you see what lines 190, 200, 220, and 230 do? You might want to experiment with the numbers in these statements at this point. Next we'll talk about color.

Color

You noticed when you ran your program that four colors appeared on the screen: the background was black, the rectangle was orange, one diagonal was green, and the other diagonal was blue.

Look at line 120 of the program:

```
120 COLOR 1
```

This command causes lines 130 to 170 to be executed using the color *orange*. Similarly, line 180

```
180 COLOR 2
```

causes the top-left to bottom-right diagonal of the X to be drawn in *light green*.

Thus, to light up a pixel, we need to specify two things:

1. Where the pixel is on the screen
2. What color we want the pixel to be

We can see from our program that the colors produced by the COLOR statements are as follows:

TABLE 3-2 Default Colors - Graphics 3, 5, 7	
COLOR 0:	BLACK (screen background)
COLOR 1:	ORANGE .
COLOR 2:	LIGHT GREEN .
COLOR 3:	BLUE .

These colors are known as the *default colors* because they are the colors we will get unless we give the computer additional instructions.

But wait! COLOR 0? We don't even have such a statement in our program! How could we see black pixels on a black background? Add the following lines to your program to see the sense of using a COLOR 0 statement:

```
240 COLOR 0
250 PLOT 1,1
260 DRAWTO 38,18
```

What happened? Did you see the top-left to bottom-right diagonal appear and then very quickly disappear? Notice that lines 250 and 260 that we added are exactly the same as lines 190 and 200. We simply redrew the line, but *in the same color as the background*! This, of course, makes the line disappear. (This is a fundamental technique of computer animation.)

Now let's experiment just a bit. If we change line 120 to

```
120 COLOR 2
```

then the rectangle will be light green instead of orange. Similarly, we can change the colors of the diagonal also. Make some experiments now on your own by changing the COLOR statements in the program.

Were you able to produce the colors you wanted in your experiments? Keep in mind that a COLOR statement determines the color that will be used until the next COLOR statement occurs in the program.

Now, you may well be thinking, "But black, orange, green, and blue aren't really my favorite colors!" Not to worry. The ATARI provides several different colors in different intensities. We will refer to a color as a *hue* and to an intensity as a *luminance*. We can use a maximum of four hues on the screen at a time. Each of these hues is controlled by a *color register* that has one of the numbers 0, 1, 2, or 4. Think of each color register as a "paint pot" that can contain different colors of paint. To put a particular color paint in a paint pot, we use the SETCOLOR command, such as

```
110 SETCOLOR 1,14,10
```

Setcolor

The 1 refers to the color register (0, 1, 2, 4); the 14 refers to the hue (a number between 0 and 15); and the 10 refers to the luminance (an even number between 0 and 14). In this particular case, the command fills paint pot #1 with yellow paint. Table 3-3 shows which hue and luminance numbers to use for some standard colors. You will want to refer to this chart frequently as you program color graphics.

TABLE 3-3   Standard Colors - Hue and Luminance Numbers

Color	Hue Number	Luminance Number
Black	0	0
White	0	14
Gray	0	6
Brown	14	0
Red	4	4
Orange	2	6
Yellow	14	10
Green	12	4
Blue	7	2
Violet	5	2
Pink	4	10
Flesh	3	12
Gold	1	6
Rust	2	0
Turquoise	10	8
Forest Green	11	2
Olive Drab	13	2
Normal Screen Blue	9	4

Now let's type in a program to see just how the SETCOLOR command works. If you still have a program in the computer, be sure to type NEW before you enter this program:

```
100 REM - Chapter 3, No. 2
110 GRAPHICS 3
120 SETCOLOR 4,0,0
130 SETCOLOR 0,4,4
140 SETCOLOR 1,0,14
150 SETCOLOR 2,7,2
160 COLOR 1
170 PLOT 5,5
180 DRAWTO 34,5
190 COLOR 2
200 PLOT 5,6
210 DRAWTO 34,6
220 COLOR 3
230 PLOT 5,7
240 DRAWTO 34,7
```

If everything worked correctly, your display was a flag of horizontal red, white, and blue stripes on a black background. The SETCOLOR statements are responsible for the colors black, red, white, and blue. Remember the paint pots? The SETCOLOR statements in lines 120 to 150 fill the four paint pots (color registers) with black (**0,0**), red (**4,4**), white (**0,14**), and blue (**7,2**) according to Table 3-3. Remember that the first number in the SETCOLOR statement is the number of the color register (4, 0, 1, 2); the second number is the hue; and the third number is the luminance. The luminance numbers are small (0, 2, 4) for dark luminances, and large (10, 12, 14) for bright luminances.

So far so good? Our four paint pots are filled with black, red, white, and blue paint—now, how do we use each paint pot? We already know about the COLOR statement. Clearly each COLOR statement must refer to one of the SETCOLOR statements. Unfortunately, the COLOR numbers are not the same as the *color register* numbers in the SETCOLOR statements. Table 3-4 shows how the SETCOLOR and COLOR numbers correspond for graphics modes 3, 5, and 7.

**TABLE 3-4   SETCOLOR and COLOR Numbers -
Graphics 3, 5, 7**

SETCOLOR Number (Color Register)	COLOR Number	Default Color (no SETCOLOR)
0	1	orange
1	2	light green
2	3	blue
4	0	black

Now see if you can match up the SETCOLOR statements with their corresponding COLOR statements. List your program on the screen, and refer to Table 3-4.

Because SETCOLOR number 0 corresponds to COLOR number 1, line 130 (SETCOLOR **0,4,4**) must correspond to line 160 (COLOR 1). That is, line 130 fills color register 0 with red (**4,4**) and line 160 tells the computer to PLOT and DRAWTO in red at lines 170 and 180. Similarly, line 140 matches line 190, and line 150 matches line 220.

But what about line 120? We have a SETCOLOR 4 statement but no matching COLOR 0 statement. Here is an important rule to remember.

**RULE: In graphics modes 3, 5, and 7, SETCOLOR
4 always determines the color of the screen
background.**

Because our command at line 120 was SETCOLOR **4,0,0**
(black), the background of our display was black. To illustrate
further, let's change the background to gold (in color) by
changing line 120 to

```
120 SETCOLOR 4,1,6
```

Refer to Table 3-3 to confirm that **1,6** are the numbers for gold.
Did it work? Did you get a red, white, and blue flag on a gold
background? If not, LIST your program and see if you can cor-
rect it.

We would like to note here that since black is the default
color for SETCOLOR 4, if we leave line 120 out of our program,
we will automatically get a black background.

Now let's see how many different flags we can make by
changing the hue and luminance numbers in the SETCOLOR
commands in lines 130, 140, and 150. First, let's change back to
a black background by changing line 120 to

```
120 SETCOLOR 4,0,0
```

Try the following changes for an orange, a yellow, and a green
flag:

```
130 SETCOLOR 0,2,6
140 SETCOLOR 1,14,10
150 SETCOLOR 2,12,4
```

If you are bored with flags with horizontal stripes, you can
change the numbers in the PLOT and DRAWTO statements to
make vertical, or even diagonal, stripes. This is a good time for
you to experiment using the different SETCOLOR numbers
from Table 3-4. If you're ambitious, perhaps you can produce an
original color masterpiece!

The Text Window

You may have noticed in the preceding graphics program that
the display always contained a rectangle at the bottom of the
screen in which the word READY and the cursor appeared. This
rectangle is known as the *text window* and provides us with a
way to combine *text* (written information) with graphics dis-
plays. Type in and run the following program to see how the
text window can be used.

```
100 REM - Chapter 3, No. 3
110 GRAPHICS 3
120 SETCOLOR 4,12,4
130 SETCOLOR 2,4,4
140 PRINT "LINE 1"
150 PRINT "LINE 2 - RED TEXT WINDOW"
160 PRINT "LINE 3 - GREEN BACKGROUND"
170 PRINT "LINE 4";
180 GOTO 180
```

Your display should be a green screen with a red text window in which is written:

```
LINE 1
LINE 2 - RED TEXT WINDOW
LINE 3 - GREEN BACKGROUND
LINE 4
```

We can see from the program and the display that

1. The text window contains 4 lines
2. SETCOLOR 2 determines the background color of the text window

Because **4,4** are the hue and luminance colors for red, line 130 (SETCOLOR **2,4,4**) produced the red text window. We know that SETCOLOR 4 determines the color of the screen background in GRAPHICS 3, so line 120 (SETCOLOR **4,12,4**) gives us a green background.

Note particularly lines 170 and 180 of the program:

```
170 PRINT "LINE 4";
180 GOTO 180
```

The semicolon (;) at the end of line 170 ensures that the cursor stays on the fourth line of the text window. Otherwise, the cursor would move to the beginning of the next line and push line 1 off the top of the window. (You can test this by removing the semicolon.) Line 180 prevents the word READY from being printed at the bottom of the window. This would also force line 1 (and line 2) off the top of the window. Think of the line "180 GOTO 180" as a way of keeping the computer from doing anything else—it is chasing its tail.

If you feel (as we do) that it is somewhat distracting to have a cursor left on the screen, you can get rid of it by adding the

following line to your program:

```
135 POKE 752,1
```

For the time being, just remember that this command gets rid of the cursor.

Now practice changing the colors of the background and text window. Put your own message in the PRINT statements in lines 140 to 170.

Perhaps you don't need a text window in a particular graphics display. We can change from a split screen (graphics window plus text window) to a full screen (graphics window only) by simply adding 16 to the graphics number. For example, instead of GRAPHICS 3, we could use GRAPHICS 19 or GRAPHICS 3+16. Try the following program as an illustration (don't forget NEW). Note that we are using the default colors, but you could add some SETCOLOR statements if you like.

```
100 REM - Chapter 3, No. 4
110 GRAPHICS 3+16
120 COLOR 1
130 PLOT 0,0
140 DRAWTO 39,0
150 DRAWTO 39,23
160 COLOR 2
170 PLOT 38,23
180 DRAWTO 0,23
190 DRAWTO 0,1
200 GOTO 200
```

Your result should be an orange and light green border around the screen. Note that because we got rid of the text window, we now have additional rows of pixels at the bottom of the screen. The number of rows depends on the graphics mode: four more rows in GRAPHICS 3, eight more rows in GRAPHICS 5, and sixteen more rows in GRAPHICS 7. Table 3-5 on the next page is an update of Table 3-1.

Text Modes

We have just looked at the three graphics modes—3, 5, and 7. The ATARI also has three *text modes*. One of these, GRAPHICS 0, is the mode that we are automatically in when we turn on the computer. We notice that we normally have a blue screen with a black border and white (very pale blue, actually) letters. All of

TABLE 3-5   Screen Formats - Graphics, 3, 5, 7 (full and split)

Graphics Mode Number	Columns (Left to Right)	Rows (Top to Bottom)	
		Full Screen	Split Screen
3	40	24	20
5	80	48	40
7	160	96	80

these conditions can be changed with SETCOLOR statements. We will not need to use COLOR statements with the text modes. Try this program:

```
100 REM - Chapter 3, No. 5
110 GRAPHICS 0
120 SETCOLOR 2,4,4
130 SETCOLOR 4,12,4
140 PRINT "GRAPHICS 0"
150 PRINT
160 PRINT "RED BACKGROUND"
170 PRINT
180 PRINT "GREEN BORDER"
190 PRINT
200 PRINT "LIGHT PINK LETTERS"
```

We hope that you got a red background with a green border and light pink letters, just as the program says! Here are some facts to remember about graphics mode 0:

1. A GRAPHICS 0 command at the beginning of the program will clear the screen.

2. SETCOLOR 2 controls the color of the background. (**4,4** is *red* in the preceding program.)

3. SETCOLOR 4 controls the color of the border. (**12,4** is *green* in the preceding program.)

4. SETCOLOR 1 controls the luminance of the characters. They are the same color as the background but brighter or darker, depending on which luminance value you use in the SETCOLOR 1 statement. You may leave the SETCOLOR 1 statement out, and usually the default value for the character luminance will be satisfactory. (It might happen

that you get something like yellow letters on a yellow background, which will be invisible! If so, add a SETCOLOR 1 statement and experiment with the luminance value.)

5. There are 24 rows of 40 characters each on the GRAPHICS 0 screen. The left margin is automatically set for the third column (column 2—we start with 0), and the right margin is set for the fortieth column (column 39), so that there are 38 characters on a line.

6. If you run a GRAPHICS 0 program that contains SET COLOR 2 and/or SETCOLOR 4 statements to change the colors of the background and/or the border, then those colors will stay on the screen until you press the SYSTEM RESET key. For example, if you LIST your program, it will be listed in the new colors. (Some programmers like to customize their ATARI screens by making them green or orange, or whatever color they find pleasing to work with.)

To see how GRAPHICS 1 and GRAPHICS 2 work, run the following program:

```
100 REM - Chapter 3, No. 6
110 GRAPHICS 1
120 SETCOLOR 4,14,0
130 SETCOLOR 0,4,10
140 SETCOLOR 1,14,10
150 SETCOLOR 2,9,4
160 POSITION 2,2
170 PRINT #6;"GRAPHICS 1"
180 POSITION 2,4
190 PRINT #6;"brown background"
200 POSITION 2,6
210 PRINT #6;"PINK LETTERS"
220 POSITION 2,8
230 PRINT #6;"yellow letters"
240 PRINT "BLUE TEXT WINDOW"
```

See what happens if you change line 110 to

```
110 GRAPHICS 2
```

Three things are immediately obvious about GRAPHICS 1 and GRAPHICS 2:

1. **There is no border on the screen.**

2. **There *is* a text window.**

3. **We use a POSITION statement to place text on the screen.**

The POSITION statement is very similar to the PLOT statement; it gives a pair of numbers that tell how many columns to the right and how many rows from the top to start printing the information in the *next* PRINT statement. For example, in the preceding program, GRAPHICS 1 is printed starting at the third column, third row (POSITION **2,2**).

GRAPHICS 1 has 20 rows of 20 characters each, plus 4 rows of text window. Thus, GRAPHICS 1 characters are the same height but twice as wide as GRAPHICS 0 characters.

GRAPHICS 2 has 10 rows of 20 characters each, plus 4 rows of text window. Thus, GRAPHICS 2 characters are the same width, but twice as high as GRAPHICS 1 characters.

Note line 170 of the preceding program:

```
170 PRINT #6;"GRAPHICS 1"
```

In this line, #6 is a code for the graphics window—we want to print GRAPHICS 1 in the graphics window, not the text window. At line 240, we do not use the code #6, so BLUE TEXT WINDOW gets printed in the text window. Remember, to print in the graphics window in graphics modes 1 and 2, you must use the code #6 after the PRINT command, followed by a semicolon (;).

We may also get rid of the text window in GRAPHICS 1 and 2 by adding 16 to the graphics mode number, just as we did in GRAPHICS 3, 5, and 7. If you don't use a text window, be sure to use the code #6 after all PRINT commands; otherwise, the display will revert to GRAPHICS 0.

If you run the preceding program again (in either GRAPHICS 1 or GRAPHICS 2), you will notice that text that was in uppercase letters in a PRINT statement appeared in pink on the screen, and text that was in lowercase letters in a PRINT statement appeared in yellow on the screen. Note that pink and yellow are the colors determined by the SETCOLOR statements in lines 130 and 140, respectively. Thus, in GRAPHICS 1 and 2, *the color of the characters is determined by the type of character* (uppercase, lowercase, inverse, etc.) *used in the* PRINT *statement*. Table 3-6 on page 38 shows the combinations.

Test this information in a program of your own. Don't feel lost if you find it a bit confusing at this time, just keep working at it.

Now you know (almost) everything about using color graphics on the ATARI. Don't be embarrassed if you find you must keep referring to the charts and examples—learning to program takes some practice!

TABLE 3-6   Character Types/Setcolor Numbers—
Graphics 1 & 2

SETCOLOR Number	Default Color	Character Type in "PRINT"	Feature Controlled by SETCOLOR
0	orange	uppercase and numbers	character
1	light green	lowercase	character
2	blue	inverse uppercase and inverse numbers	character, text window color
3	red	inverse lowercase	character
4	black	—	background color

Sound

The SOUND command for the ATARI is somewhat similar to the SETCOLOR command. Just as we have four color registers (paint pots), we have four *voices*, numbered 0, 1, 2, and 3. That means that we can have four separate sound channels playing simultaneously. Corresponding to hue for SETCOLOR is *pitch* (tone) for SOUND, and corresponding to luminance is *volume control*. SOUND has an additional characteristic (parameter); namely, distortion. This is the quality that produces special sound effects for games, etc. We can test the SOUND statement by typing in the following command—without a line number—and pressing the RETURN key. Press the SYSTEM RESET key to turn the sound off.

```
SOUND 0,121,10,8
```

You should have heard a tone that was approximately middle C. Let's consider the separate numbers in this SOUND statement:

0   Refers to the voice (0,1,2,3)

121 Refers to the pitch (a number between 0 and 255, where 0 is the highest tone, and 255 is the lowest tone)

10   Refers to the distortion (an even number between 0 and 14. 10 is a pure, or musical, tone.)

8   Refers to volume control (a number between 1 and 15, where 1 is barely audible and 15 is loud.)

Try two voices together:

```
SOUND 0,121,10,8
SOUND 1,60,10,8
```

You should have heard middle C and C above middle C. Now try some distortion:

```
SOUND 0,121,8,8
SOUND 1,60,2,8
```

You describe what you heard!

Here is a program to play some familiar notes. It contains some new commands, which we will discuss later. You can probably guess how they work. Play close attention to the punctuation and notice that line 120 is the same as line 110 (we want to play note 121 twice), line 140 is the same as line 130, etc. To duplicate a line without retyping the entire line, simply move the cursor to the line number and type the new line number. Press the RETURN key, and the line will be duplicated. Type LIST and press RETURN to verify the duplication.

```
100 REM - Chapter 3, No. 7
110 SOUND 0,121,10,10:GOSUB 300
120 SOUND 0,121,10,10:GOSUB 300
130 SOUND 0,96,10,10:GOSUB 300
140 SOUND 0,96,10,10:GOSUB 300
150 SOUND 0,81,10,10:GOSUB 300
160 SOUND 0,81,10,10:GOSUB 300
170 SOUND 0,96,10,10:GOSUB 300
180 STOP
300 FOR HOLD=1 TO 100:NEXT HOLD
310 FOR OFF=1 TO 10
320 SOUND 0,0,0,0
330 NEXT OFF
340 RETURN
```

We'll learn some easier ways to program music later. Let's look at some features of the program. Line 300 determines the *duration* of each note played. Think of the computer as counting to 100 while it plays a note. In this program, each note has the same duration. Lines 310 to 330 turn the note off while the computer counts to 10. Without this *interval*, the notes slur together, a technique we often need to use in programming music or sound effects. Line 340 tells the computer to go back and play the next note. You may wish to see what happens if you change the length

of the duration and interval in lines 300 and 310.

Try this sound effects program that simulates a tug boat (Thanks to Tom Rowley!):

```
100 REM - Chapter 3, No. 8
110 SOUND 0,250,12,10
120 SOUND 1,243,10,10
130 SOUND 2,29,10,8
140 FOR OFF=1 TO 8
150 SOUND 0,0,0,0
160 SOUND 1,0,0,0
170 SOUND 2,0,0,0
180 NEXT OFF
190 GOTO 100
```

This tug keeps chugging until you press the BREAK key or the SYSTEM RESET key.

The REMARK Statement

As our programs get longer and more complex, you can see that it becomes more and more difficult to remember what part of the program does what. Fortunately, we can put comments directly into a program by using the REMARK statement. We have already been using REMARK (abbreviated, REM) statements to identify each of the programs in this chapter by number. A REM statement does not affect the running of a program; it is just for our information. It is part of the *documentation* of the program; that is, the set of explanatory information that accompanies a program. Try this simple example of a program with REMARK statements.

```
100 REM *** Chapter 3, No. 9     ***
110 GRAPHICS 2+16
120 REM *** brown background     ***
130 SETCOLOR 4,14,2
140 REM *** yellow = upper case ***
150 SETCOLOR 0,14,10
160 REM *** pink   = lower case ***
170 SETCOLOR 1,4,10
180 POSITION 6,3
190 PRINT #6;"PROGRAM"
200 POSITION 7,5
210 PRINT #6;"TITLE"
220 POSITION 4,8
230 PRINT #6;"by john doe"
240 REM *** hold text on screen ***
250 GOTO 250
```

We close this chapter with a long program that combines graphics and sound to provide an animated figure. There are many REMARK statements to tell you what's happening. Note that in several places we have put two related commands on one line, separated by a colon (:).

```
100 REM *** Chapter 3, No. 10        ***
101 REM *** Singing Practice         ***
110 GRAPHICS 3+16
120 REM *** set up colors            ***
130 SETCOLOR 4,14,2:REM color 0=brown *
140 SETCOLOR 0,0,14:REM color 1=white *
150 SETCOLOR 1,4,4:REM .color 2=red   *
160 SETCOLOR 2,0,0:REM .color 3=black *
170 REM *** draw eyeballs            ***
180 COLOR 1:PLOT 14,7:DRAWTO 16,7
190 PLOT 23,7:DRAWTO 25,7:PLOT 14,8
200 PLOT 15,8:PLOT 23,8:PLOT 24,8
230 REM *** draw pupils              ***
240 COLOR 3:PLOT 16,8:PLOT 25,8
250 REM *** draw closed mouth         ***
260 COLOR 2:PLOT 17,13:DRAWTO 22,13
270 PLOT 17,14:DRAWTO 22,14
280 REM *** wait a moment            ***
290 FOR WAIT=1 TO 500:NEXT WAIT
300 REM *** pick a note              ***
310 SOUND 0,60,10,10:GOSUB 400
320 SOUND 0,47,10,10:GOSUB 400
330 SOUND 0,40,10,10:GOSUB 400
340 SOUND 0,29,10,10:GOSUB 400
350 SOUND 0,40,10,10:GOSUB 400
360 SOUND 0,47,10,10:GOSUB 400
370 SOUND 0,60,10,10:GOSUB 400
380 GOTO 290:REM *** do it again     ***
390 REM *** erase closed mouth       ***
400 COLOR 0:PLOT 17,13:DRAWTO 22,13
410 PLOT 17,14:DRAWTO 22,14
420 REM *** draw open mouth          ***
430 COLOR 2:PLOT 18,12:DRAWTO 21,12
440 DRAWTO 21,15:DRAWTO 18,15
450 DRAWTO 18,12
460 REM *** move pupils              ***
470 COLOR 1:PLOT 16,8:PLOT 25,8
480 COLOR 3:PLOT 16,7:PLOT 25,7
490 REM *** hold note                ***
500 FOR HOLD=1 TO 100:NEXT HOLD
510 REM *** erase open mouth         ***
520 COLOR 0:PLOT 18,12:DRAWTO 21,12
530 DRAWTO 21,15:DRAWTO 18,15
```

```
540 DRAWTO 18,12
550 REM *** move pupils                  ***
560 COLOR 1:PLOT 16,7:PLOT 25,7
570 COLOR 3:PLOT 16,8:PLOT 25,8
580 REM *** draw closed mouth            ***
590 COLOR 2:PLOT 17,13:DRAWTO 22,13
600 PLOT 17,14:DRAWTO 22,14
610 REM *** turn off note                ***
620 FOR OFF=1 TO 20
630 SOUND 0,0,0,0
640 NEXT OFF
650 RETURN
```

Whew! If nothing else, you got some good typing practice. Note that the technique for erasing pixels is to redraw them in the background color. Then if you draw them somewhere else in the original color, you get the effect of animation. You might want to modify this program to produce your own animated picture as a project.

EXERCISES

1. The number of the error message that means "Cursor out of range"

    is _____ .

2. [( ) True   ( ) False]   The COLOR statement determines which
    color will be stored in a color register.

3. Of graphics modes 3, 5, and 7, graphics mode _____ has the
    most pixels.

4. In graphics modes 3, 5, and 7, the upper left corner of the screen

    is numbered _____ , _____ .

5. In graphics mode 7, the lower right corner of the GRAPHICS

    WINDOW is numbered _____ , _____ .

6. [( ) True   ( ) False]   The text window in GRAPHICS 7 has more
    lines than the text window in GRAPHICS 3.

7. GRAPHICS 5 (with a text window) has a *total* of _____ pixels
    in the graphics window.

8. The BASIC command to light up a single pixel on the screen is

    _____ .

9. The default color for COLOR 1 is _____ .

10. The command to produce a pink background in GRAPHICS 3 would

    be _____ .

11. [( ) True   ( ) False]   The command `PLOT 10,5:DRAWTO 10,5`
will produce an error message.

12. In graphics modes 3, 5, and 7, how many colors can be placed on the

screen at one time? _____ .

13. In graphics modes 3, 5, and 7, the color of the background is deter-

mined by SETCOLOR _____ .

14. In graphics mode 7 (with a text window) there are _____ times
as many pixels as in graphics mode 5 (with a text window).

15. [( ) True   ( ) False]   The command `SETCOLOR 0,14,10` will
produce a yellow background in GRAPHICS 3.

16. Which of the following statements is true?

( )   If a DRAWTO statement in GRAPHICS 3 doesn't produce
an error 141 message, then it won't produce one in GRAPHICS 5,
either.

( )   If a DRAWTO statement in GRAPHICS 5 doesn't produce an
error 141 message, then it won't produce one in GRAPHICS 3,
either.

17. To eliminate the text window in graphics modes 3, 5, and 7, we add

_____ to the number of the graphics mode in the GRAPHICS
statement.

18. [( ) True   ( ) False]   The statement `SOUND 0,62,20,10` will
produce an error message.

19. To print HELP! on the GRAPHICS 1 screen at position 5,5, the

PRINT command would be _____ .

20. [( ) True   ( ) False]   The SOUND statement will work only
with GRAPHICS 0.

21. A screen that includes a graphics window and a text window is

called a _____ screen.

22. [( ) True   ( ) False]   The command **SOUND 0,121,6,8** will produce note middle C.

23. Which of the following statements is true?

( )   The command **SOUND 3,60,10,10** produces a louder sound than the command **SOUND 0,60,10,10.**

( )   The command **SOUND 2,30,10,8** produces a higher pitched sound than the command **SOUND 0,50,10,8.**

24. There are _____ sound registers (channels).

25. Write the statement that will turn off the sound generated by the

command **SOUND 2,87,10,6** _____ .

26. [( ) True   ( ) False]   To duplicate a line in a program, we can type over the line number and press the RETURN key.

27. To add a comment to a program, we can use a _____ statement.

28. Write a GRAPHICS 3 program that will draw an orange square 15 pixels long on each side anywhere on the screen.

29. Write a GRAPHICS 2 program that will print in red letters on a yellow background this message:

RED

ON

YELLOW

4

Assignment (LET) Statements

Much of the power of a computer lies in its ability to perform the same computation over and over but on different sets of numbers. To do this, the computer uses *variables*. In this chapter we will look at the rules for using numerical and alphabetical (string) variables. At the end of the chapter, we will discuss saving your programs on tape and disk.

The computer has the capacity to store information and retrieve it at a later time. This storage capacity is referred to as *memory*. The unit of measure for computer memory is the *byte*, which is the amount of memory needed to store one alphabetic or numeric character. Your ATARI may have between 5000 and 48,000 bytes of memory. One thousand bytes of memory is known as a *kilo byte*, or simply a K. Thus, 32K memory means 32,000 bytes, or characters, of memory. (Actually, 32K means 32,768 bytes—2 raised to the 15th power.)

The computer memory, of course, is a combination of electronic parts, but it is useful for us to think of the memory as a set of *boxes* in which we can *store* numbers and letters. (We sometimes refer to memory as storage.) In order to know which box a particular number or set of letters is stored in, we must attach a label to the box, just as a person attaches his name to his mailbox. These labels are known as *variables*.

Numeric Variables

A *numeric variable* is a label for a memory location (one of the "boxes" in the computer memory) that can hold a number. In ATARI BASIC, a numeric variable name can be any combination of uppercase letters and numbers that satisfies these two simple rules:

1. **The first character must be a letter**
2. **The length does not exceed 120 characters**

Here are some possible numeric variable names:

JOJO	**R2D2**	**NEWX**	**OLDX**	**BIGBOY**
LENGTH	**AREA**	**A5**	**Z67**	**JAN25**
HUE	**LUM**	**PITCH**	**DISTORT**	**VOLUME**
HEIGHT	**LASTTESTGRADE**			**AGE7**

Here is a program using variables that does the same thing as the first program in Chapter 2:

```
100 REM - Chapter 4, No. 1
110 PRINT "ADD TWO NUMBERS"
120 PRINT
130 LET A=5.3
140 LET B=9.8
150 PRINT A;" + ";B;" = ";A+B
```

Did your run look like this?

```
RUN
ADD TWO NUMBERS

5.3 + 9.8 = 15.1

READY
■
```

Let us now consider the program in detail. We identified the variables at lines 130 and 140. Look at line 130:

```
130 LET A=5.3
```

The sense of this statement is: LET the box labeled A contain the number 5.3. Similarly, line 140 means: LET the box labeled B contain the number 9.8. Each of these two statements did two things:

1. It named a variable (**A, B**)
2. It gave each variable a *value* (5.3, 9.8)

Now look at line 150:

```
150 PRINT A;" + ";B;" = ";A+B
```

The computer prints the contents of *the box with label* "A"; that is, variable A. Then it prints " + ". Remember that anything that is enclosed in quotation marks is printed exactly as it appears inside the quotes. Recall also that the semicolon (;) produces *compact spacing*: no space is printed between items separated by a semicolon. This is why we needed a space on either side of the " + " in line 150. After the " + ", the contents of variable B are printed; then " = "; then the *sum of the contents of variables* A and B (5.3+9.8, OR 15.1).

To see the real power of the LET statement, add the following lines to your program. The display will keep on going to infinity (if you have that much time), so to stop it, hold down the CTRL key, and press the 1 key. To start it again, do the same thing. An alternative method is to press the BREAK key.

```
101 REM - Addition to Chapter 4, No. 1
160 LET A=A+1
170 LET B=B+2
180 GOTO 150
```

If you stopped the display quickly, you could see that variable A was increased by 1, and variable B was increased by 2 each time and then the new values and the sum were printed. The sense of line 160 is: Increase the value of variable A by 1, and then store the result back in variable A. Read the equals sign in LET statements as be replaced by, or is replaced by, and they will make more sense. (Algebra students know that no real number satisfies the equation A = A + 1.)

If we use a variable in a LET statement and we have not previously given a value to the variable, the value will automatically be set to zero (0). Here is an example:

```
100 REM - Chapter 4, No. 2
110 LET B=5
120 PRINT "A = ";A
130 PRINT "B = ";B
140 PRINT "A + B = ";A+B
```

Here is an example of a program with LET statements. On the right we show what is in each memory location as the program is run.

MEMORY			
M	N	Q	R

```
100 REM - Chapter 4, No. 3
110 LET M=7 ─────────────────►7
120 LET N=23 ─────────────────►23
130 LET Q=M*N ─────────────────►161
140 LET R=M+N-5 ─────────────────►25
150 LET R=R/5 ─────────────────► 5
160 PRINT M,N,Q,R
```

The display should look like this:

```
RUN
7            23          161         5

READY
■
```

The word LET may be omitted in LET statements, as the following program shows. See if you can correctly guess the display before you run the program.

```
100 REM - Chapter 4, No. 4
110 A=6
120 B=8
130 A=(A+B)/7
140 B=A+B/2
150 PRINT "A = ";A
160 PRINT "B = ";B
```

Did you guess that the display is like this?

```
RUN
A = 2
B = 6

READY
■
```

Try this program to draw a line slowly in GRAPHICS 5:

```
100 REM - Chapter 4, No. 5
110 GRAPHICS 5+16
120 COLOR 1
130 COL=16
140 ROW=0
150 PLOT COL,ROW
160 SOUND 0,4*COL,10,10
170 FOR WAIT=1 TO 100:NEXT WAIT
180 COL=COL+1
190 ROW=ROW+1
200 IF ROW<48 THEN 150
210 SOUND 0,0,0,0
220 GOTO 220
```

Line 200 provides a way of continuing until the line reaches the bottom of the screen. We will discuss IF...THEN statements in detail in Chapter 7.

Notice that the three variable names we used in this program (COL, ROW, and WAIT) actually suggest the items they represent. COL tells the COLumn of the pixel to be lighted, and ROW tells the row. WAIT indicates the pause between lighting two successive points on the line. As you write your own programs, try to choose meaningful variable names, so that others can follow your code more easily.

Refer once more to line 160 of the program:

```
160 SOUND 0,4*COL,10,10
```

Here the pitch number is actually a *variable expression* (4*COL). It is always 4 times the current value of COL. Because the value of COL increases from 16 to 63 (1 is added 47 times), the value of 4*COL increases from 64 (4 times 16) to 252 (4 times 63). You will recall that the range of possible numbers for the pitch in a SOUND statement is from 0 to 255, so 4*COL gives a permissible set of values. You might want to try some other possibilities instead of 4*COL at line 160: COL, 2*COL, or even 6*COL (you'll be surprised at the results of the last one!).

So far we have only discussed how to store *numbers* in memory locations in the computer, but we also know that computers must be able to store names and other *alphabetic* information. To do this, we need another kind of variable.

String Variables

A *string variable* is a label for a memory location that can hold alphabetic information (a string of letters). A string variable name is like a numeric variable name except for one thing: The last character of the name is a dollar sign ($). Here are some possible string variable names. Compare them to the list in the section "Numeric Variables."

JOJO$	**R2D2$**	**NEWX$**	**OLDX$**	**BIGBOY$**
LENGTH$	**AREA$**	**A5$**	**Z67$**	**JAN25$**
HEIGHT$	**LASTTESTGRADE$**			

The *strings* (letters, numbers, or punctuation marks treated as characters) that we want to use in a program will not always have the same number of characters—a name may be 10 letters long, but a street address may be 60 letters long. Therefore, we must tell the computer the maximum number of characters that we intend to put in a string variable. We do this at the beginning of the program in a DIMENSION (abbreviated DIM) statement.

Here is a program that shows the use of the DIM statement and string variables:

```
100 REM - Chapter 4, No. 6
110 DIM NAME$(20),DATE$(9)
120 NAME$="JOHNNY JONES"
130 DATE$="25 SEP 82"
140 PRINT NAME$;
150 PRINT " did this on ";
160 PRINT DATE$
```

You should get this display:

```
RUN
JOHNNY JONES did this on 25 SEP 82

READY
■
```

Note the following in the preceding program:

1. **Two string variables are dimensioned at line 110: NAME$ and DATE$. Immediately following the string variable name we place in parentheses the maximum number of characters the variable will hold. (The variable may hold this number of characters, or any lesser number of characters.)**

2. **At lines 120 and 130, the strings to be stored in NAME$ and DATE$ are enclosed in quotation marks.**

3. **The semicolon (;) at the end of line 140 is an instruction to print the next item (line 150) starting at the *next column on the same line*, instead of the first column of the next line. The semicolon at the end of line 150 serves the same purpose.**

Put your own name in the quotes in line 120 and *run* the program again. See what happens if you remove the semicolons at the ends of lines 140 and 150. Experiment, experiment, experiment!

We can print only part of a string—see if you can see how the following program works:

```
100 REM - Chapter 4, No. 7
110 DIM A$(25)
120 A$="THE COMPUTERS ARE FUN!"
130 PRINT A$(8,10);" ";A$(1,3);A$(4,5);
140 PRINT A$(15,15);A$(1,1);" ";
150 PRINT A$(6,6);A$(9,10);A$(22,22)
```

Were you surprised at the display? The numbers in the parentheses after the string variable name give the numbers of the first character and the last character of the string. For example, if

**A$="VARIABLE", then A$(2,5)="ARIA",
and A$(5,8)="ABLE"**

Try one more program with numeric and string variables:

```
100 REM - Chapter 4, No. 8
110 DIM NAME$(10)
120 NAME$="LUDWIG"
130 HOURS=23
140 RATE=2.85
150 WAGES=HOURS*RATE
160 PRINT "NAME","HOURS","RATE","WAGES"
170 PRINT
180 PRINT
```

Now you need to find out how to save your programs on cassette tapes and diskettes.

Saving Programs on Cassette Tapes

In Chapter 1 you learned how to load programs from a cassette tape into the computer. The steps necessary to save a program from the computer onto a cassette tape are very similar. First, be sure your ATARI Program Recorder is correctly attached to the computer:

1. **Plug the recorder data cord into the jack labeled PERIPHERAL on the side or back panel of the ATARI console.**

2. **Plug the recorder power cord into an ordinary 110V wall socket or transformer.**

Now, be sure that your program on the computer is working the way you want it to, and then follow these steps:

1. **Insert your blank cassette tape into the program recorder with the recording surface toward you and the label right side up, so that you can read it.**

2. **Push REWIND on the recorder and wait until the tape stops.**

3. **Push the tape counter reset button until it reads 000, then push STOP EJECT once.**

4. **On the computer keyboard, type CSAVE and then press the RETURN key. You will hear two beeps.**

5. **Push RECORD and PLAY simultaneously on the program recorder. Now push the RETURN key on the computer keyboard again. You will hear a high-pitched whine for about 18 seconds, then a "drilling" sound. When your program has been copied, the recorder will stop and the word READY will appear on the screen.**

6. **Push STOP on the program recorder.**

7. **It is always good programming technique to create a *backup copy* of each program you wish to save. ATARI recommends that you store one program on each cassette and keep a backup on a separate cassette. To make the backup copy, simply repeat steps 1-6 with a new cassette.**

8. **Finally, write the name and tape counter number of your new program on each cassette label and also on a page that you keep with your other programming documentation.**

It would be a good idea now to practice saving some short programs on tape until you can perform the entire operation without referring to these notes.

Saving Programs on Diskettes

You will remember from Chapter 1 that in order to load a program from a diskette, you must have the ATARI Disk Drive attached to your computer (with at least 16K RAM), and you

must also load DOS (Disk Operating System). You must do the same things to *save* a program on a diskette. Here is a review of the steps to attach the disk drive to the computer:

1. **Plug one end of a data cord into the jack labeled PERIPHERAL on the side or back panel of the ATARI console. Plug the other end into *either* of the jacks labeled I/O CONNECTORS on the back of the disk drive unit. NOTE: If you also want to use the recorder, you may plug its data cord into the other I/O CONNECTOR jack.**

2. **Plug the small plug on the AC power adaptor into the jack labeled POWER IN on the back of the disk drive unit and the other end into an ordinary 110V wall socket or transformer.**

To load DOS, do the following:

1. **Press the PWR. ON/OFF switch on the front of the disk drive. The two red lights will glow, the drive will whir for about 7 seconds, and then the *upper* red light will go out. The *lower* light will still glow.**

2. **Open the door in the front of the disk drive.**

3. **Insert a diskette that contains DOS with the label facing up and the notch on the left. Be careful not to touch the recording surface of the disk through the openings in the black protective jacket. Close the door of the disk drive.**

4. **Turn the computer console power switch ON. The disk drive will whir for a few seconds, then the word READY will appear on the screen. You are now ready to type in a program and then save it on a diskette. NOTE: If you need to *format* a diskette, please refer to Appendix 3.**

5. **Remove the DOS diskette from the disk drive. It is a good idea not to use the DOS diskette for anything but loading DOS. You should *write protect* your DOS diskette (and a backup copy) by covering the notches with the aluminum stickers provided with the diskettes.**

Now let's assume that you have typed your program into the computer, run it, corrected any errors, and are now ready to store it on a diskette. Unlike the recorder, the disk drive gives us the opportunity of giving names to our programs. Information stored on a diskette is referred to as a *file* (program file or data file), so we refer to *filenames*. Here are the rules for filenames:

1. **The maximum length is eight (8) characters.**

2. **The only characters that can be used are the letters A through Z, and the numerals 0 through 9.**

3. The first character is *always* an alphabetic character (A, B, C, etc.).

4. The filenames DOS.SYS, DUP.SYS, AUTORUN.SYS, and MEM.SAV are reserved for the DOS diskette.

In addition to eight characters in a filename, we can add an *extender* of not more than three alphabetic or numeric characters preceded by a period (.). Here are some examples of legal and illegal filenames with and without extenders. See if you can tell which rule each illegal filename violates:

Legal	Illegal
FILENAME.JOE	FILE #2.JOE
R2D2	R 2D2
RAIDERS.GAM	3RAIDERS.GAM
USFLAG.GRF	5,STATES.GRF
DEUTSCHL.AND	BYE-BYE

(If you are sharing diskettes with others, you may want to use your initials as extenders for identification purposes.) Let's suppose that you have picked the filename GOODWORK.ABC for your program. Follow these steps to save it on diskette:

1. Press the PWR. ON/OFF switch on the front of the disk drive.

2. When the upper red light goes out, open the door of the disk drive, insert your diskette (label up, notch to the left), and then close the door of the disk drive.

3. Type SAVE "D:GOODWORK.ABC" and press the RETURN key. The disk drive will whir for a few seconds, then READY will appear on the screen. Your program is saved on the diskette. Remove your diskette from the disk drive and turn the disk drive off.

4. Make a backup copy of your program by repeating steps 1-3 with a different diskette.

Let's look at the command we typed in step 3 more closely:

```
"D:GOODWORK.ABC"
```

The entire command is known as the *file specification* (or *filespec* for short). It has several parts as shown in Table 4-1.

If you are lucky enough to have more than one disk drive attached to your computer, you will have to specify the device number (1, 2, 3, 4) immediately after the device code (D1:, D2:, D3:, D4:). You will also have to set the DRIVE CODE SWITCH

TABLE 4-1   File Specification (Filespec) Breakdown

D	Device Code (D for "Disk")
:	Required Colon
GOODWORK	Filename (maximum 8 characters); 1st is alphabetic
.	Period required as separator if extender is used.
ABC	Extender (optional): 1-3 alphabetic/numeric characters

on the back of the disk drive. Please refer to the ATARI DISK DRIVE OPERATOR'S MANUAL.

The same filespec that is used to SAVE a program is also used to LOAD the program. The filespec is always enclosed in quotation marks. Here are some sample SAVE and LOAD commands for diskettes:

To save a program	*To load a program*
SAVE "D:GOODWORK.ABC"	LOAD "D:GOODWORK.ABC"
SAVE "D:TESTING"	LOAD "D:TESTING"
SAVE "D:MX80.EPS"	LOAD "D:MX80.EPS"

We close this chapter with an important warning.

WARNING: Never open the door of the disk drive if the upper red light is on!

EXERCISES

1. The unit of measure for computer memory is the _____ .

2. [( ) True ( ) False]   The computer memory is actually an array of microscopic boxes.

3. A variable is a ( ) box ( ) chip ( ) label for a memory location.

4. The first character of a numeric variable name must be a _____ .

5. The statement LET VAR=5 will assign the value _____ to the variable _____ .

6. [( ) True ( ) False]   Numeric variable names are limited to a length of 32 characters.

7. Write the printout for the one-line program

   200 PRINT A+5

   _____

8. Write the printout for the following program:

   100 A=21
   110 B=8
   120 PRINT (A+11)/8-4   _____

9. 8K means not exactly 8000 bytes, but _____ bytes.

10. [( ) True ( ) False]   A numeric variable can be changed to a string variable by adding a dollar sign to the end of the variable name.

11. A string variable may hold
    ( )   numerals
    ( )   letters
    ( )   punctuation marks
    ( )   all of the above

12. What statement must *always* precede the first use of a string

    variable in a program? _____

13. If NAME$="UNITED STATES",   then   NAME$(4,6)   =

    _____

14. [( ) True   ( ) False]   130 NAME$=NAME$+1   is a legiti-
    mate BASIC statement.

15. Write the printout for the following program:

    ```
    100 DIM A$(10)
    110 A$="E.T. UPMOC"
    120 PRINT A$(1,5);" LIVES."    _____
    ```

16. Which part of the statement   170 LET NAME$="BRYAN"   may

    be omitted? _____

17. [( ) True   ( ) False]   Both of the following one-line programs
    have exactly the same printout.

    ```
    100 DIM AGE$(2):AGE$="15":PRINT AGE$
    ```

    ```
    100 AGE=15:PRINT AGE
    ```

18. Check all of the legitimate string variable names:
    ( )   3P0$
    ( )   PI3.14$
    ( )   ADDRESS
    ( )   BACK-SPACE$
    ( )   S$
    ( )   "NAME"
    ( )   THISISONEOFTHELONGESTSTRINGVARIABLENAMESEVER$

19. Write the printout for the following program:

```
100 DIM NAME$(10)
110 NAME$="THEOPHRASTUS BOMBASTUS von HOHENHEIM"
120 PRINT NAME$;" IS LONG DEAD."
```

20. The keyboard command SAVE "D:BIGWORK" could be used to store a program on ( ) cassette tape   ( ) diskette.

21. [( ) True   ( ) False] 180 AGE+1=AGE is a valid command.

22. For each of the following programs, show the contents of each memory location as the program is run, as in the program Chapter 4, No. 3.

a.
```
100 REM - EXERCISE 22 A)
110 LET A=5
120 LET B=A+2
130 LET C=A*B
140 LET C=C-A
150 LET D=(C-2)/B
160 PRINT A,B,C,D
```

MEMORY			
A	B	C	D

b.
```
100 REM - EXERCISE 22 B)
110 LET A1=2
120 LET B2=6
130 LET A1=A1+B2
140 LET B2=A1/4
150 LET A1=A1/B2
160 LET B2=(A1*A1)+(B2*B2)
170 PRINT A1,B2
```

MEMORY	
A1	B2

23. Write (in the space on the right below) an *original* program that will produce the printout below. Have your teacher check that you can save the program on cassette tape and on diskette, and then load it back into the computer.

```
RUN

AREA of a CIRCLE
-----------------

RADIUS      AREA
------      ------
    5        25 Pi
   12       144 Pi

READY
■
```

( ) save on cassette   ( ) save on diskette

5

The INPUT Statement

So far, our programs with variables have had this characteristic: If we wanted to make any changes to the values of the variables, we had to change part (or parts) of the program and then run it again. In this chapter, we are going to learn a way to change the values of the variables *as we run the program.* Such a type of program is called *interactive* because we interact with the computer as the program runs.

Let's type in again a program from Chapter 4, "Chapter 4, No. 1" where we first used variables:

```
100 REM - Chapter 5, No. 1
110 PRINT "ADD TWO NUMBERS"
120 PRINT
130 LET A=5.3
140 LET B=9.8
150 PRINT A;" + ";B;" = ";A+B
```

What if we wanted to change the values of A and B each time we ran the program? We could retype lines 140 and 150, but that would be inefficient and tedious. A better way would be to use the INPUT statement.

Add these lines to your program. *Don't* type NEW!

```
110 REM - Chapter 5, No. 2
112 REM - Addition to No. 1
```

```
115 GRAPHICS O:PRINT
125 PRINT "TYPE A VALUE FOR A"
130 INPUT A
135 PRINT "TYPE A VALUE FOR B"
140 INPUT B
155 PRINT
```

When you run the program, be sure to press the RETURN key after you type in a response to a question.

Do you see that you could find the sums of many pairs of numbers in a short time this way? The INPUT statement allows us to give a value for a variable from the keyboard as the program runs. Our cue to make a response is the question mark that appears at the left edge of the screen. Do you see what lines 125 and 135 do? Right! They print instructions so that we know what to type when the question mark appears. Such an instruction in a PRINT statement is called a *prompt*. In all of the programs that you write, follow this rule.

RULE: Precede all INPUT statements with a prompt.

When you are finished experimenting with the previous program, type NEW and try this program:

```
100 REM - chapter 5, No. 3
110 GRAPHICS 3
120 PRINT "WHAT COLOR NUMBER (1,2,3)"
130 INPUT CN
140 COLOR CN
150 PLOT 1,1:DRAWTO 38,1:DRAWTO 38,18
160 DRAWTO 1,18:DRAWTO 1,1
```

See what happens if you input a number other than 1, 2, or 3. Can you guess why 1, 5, 9, 13, and 17 all produce an orange rectangle? How about 2, 6, 10, 14, and 18? Etc., etc., etc.! Experiment as much as possible with each program.

If you're satisfied that you understand the program above, then type NEW and try this one:

```
100 REM - Chapter 5, No. 4
110 GRAPHICS O:PRINT
120 PRINT "TONE TESTER"
```

```
130 PRINT
140 PRINT "PITCH VALUE (0-255)"
150 INPUT PITCH
160 PRINT "DISTORTION VALUE (0-14 EVEN)"
170 INPUT DISTORT
180 SOUND 0,PITCH,DISTORT,10
190 FOR HOLD=1 TO 500:NEXT HOLD
```

If you find some pitch and distortion values that produce good sound effects (laser, tugboat, dental drill, jet, etc.) write them down in your notebook for possible future use.

Did you remember that the GRAPHICS 0 command at the beginning of a program will clear the screen? A PRINT statement after the GRAPHICS 0 statement will give us a little border at the top of the screen. We should always attempt to make our displays look clean by clearing the screen at the beginning and by providing borders on all sides of printouts. A few sections further we will discuss the effective use of prompts.

Line 190 in the last program is necessary to keep the tone sounding; otherwise, as soon as READY is printed, the sound channels are turned off. You can vary the duration of the sound by changing the number after the word TO in line 190.

We are not limited to numeric inputs; type in and run the following program with a string input:

String (Alphabetic) Input

```
100 REM - Chapter 5, No. 5
110 DIM NAME$(10)
120 GRAPHICS 2
130 PRINT "WHAT IS YOUR FIRST NAME"
140 INPUT NAME$
150 PRINT "HOW OLD ARE YOU THIS YEAR"
160 INPUT AGE
170 POSITION 2,2
180 PRINT #6;NAME$;","
190 POSITION 2,4
200 PRINT #6;"YOU WERE BORN IN"
210 POSITION 2,6
220 PRINT #6;1982-AGE;"."
```

Try giving your age as -10 once. Hmm! Born in 1992! Well, obviously we need ways to handle inappropriate inputs, but for the time being, let's assume that the user (the person who is

typing in the responses) is a nice, upstanding person who doesn't play jokes.

Try two more experiments:

1. Give your name as 35.63
2. Give your age as MAY 29, 1967

Your results should have confirmed this rule.

> **RULE:** **Numeric variables may hold only numbers, but string variables may hold any combination of letters, numerals, and punctuation marks. In string variables, numerals are treated as symbols, not as numbers.**

In general, if the user does not match his input to the type called for in the INPUT statement(s) in the program, an error message will result, just as in experiment (2) above.

Input Error Messages

If you performed experiment (2) above properly, you got the following error message:

```
ERROR-    8 AT LINE 160
```

ERROR 8 means that the input did not match the type called for in the program. In this case, we tried to put a string (MAY 29, 1967) into a numeric variable (AGE). On the other hand, it was alright to input 35.63 as the name because NAME$ is a string variable and can hold any type of information.

Input Variations

In the preceding examples of the INPUT statement, the question mark cue for the user's response was always printed at the beginning of the line following the prompt. The question mark may also follow the prompt on the same line, as the following two programs demonstrate. Try typing the prompts in inverse video:

```
100 REM - Chapter 5, No. 6
110 GRAPHICS 0:PRINT
120 PRINT "This program will calculate"
130 PRINT "the area of a RECTANGLE if you"
140 PRINT "give its LENGTH and WIDTH."
150 PRINT
160 PRINT "Length ";:INPUT LENGTH
```

```
170 PRINT
180 PRINT "Width   ";:INPUT WIDTH
190 PRINT:PRINT
200 PRINT "LENGTH","WIDTH","AREA"
210 PRINT "------","-----","----"
220 PRINT LENGTH,WIDTH,LENGTH*WIDTH
230 PRINT:PRINT:PRINT:PRINT
```

The semicolon after the prompt keeps the input cue on the same
line. The next program uses two graphics modes:

```
100 REM - Chapter 5, No. 7
110 REM - "The Sleeper"
120 DIM NAME$(10)
130 GRAPHICS 0:PRINT
140 PRINT "What is your name ";:INPUT NAME$
150 PRINT
160 PRINT "What is your age  ";:INPUT AGE
170 PRINT
180 PRINT "How many hours do you sleep "
190 PRINT "each night (on the average) ";:INPUT HOURS
200 HOURSLIVED=AGE*24*365
210 HOURSSLEPT=AGE*HOURS*365
220 GRAPHICS 1+16
230 POSITION 1,4
240 PRINT #6;"WELL, ";NAME$;","
250 POSITION 3,8
260 PRINT #6;"you have lived"
270 POSITION 3,10
280 PRINT #6;HOURSLIVED
290 POSITION 3,12
300 PRINT #6;"hours and spent"
310 POSITION 3,14
320 PRINT #6;HOURSSLEPT
330 POSITION 3,16
340 PRINT #6;"of them sleeping!"
350 GOTO 350
```

If you are satisfied that you understand the preceding two
programs, then try this very short program to see how multiple
inputs work:

```
100 REM - Chapter 5, No. 8
110 PRINT "Type in three numbers;"
120 PRINT "for example,  5,12,13"
130 INPUT A,B,C
140 PRINT A,B,C
```

There are several ways that you can type in the three num-
bers when you run the program. Suppose you choose 5, 12, and 13
for the three numbers. Any of the following six responses will
work.

?5,12,13   ?5,12   ?5
                            ?13   ?12
                                                       ?13

?5   ?5,12,13,14,   ?5,12
?12,13   ?13,OOPS

Obviously, the last two inputs would be errors on your part, but they both would work! The computer waits until it has received three *usable* inputs, then ignores anything that is left over.

String inputs present a slightly different problem. If an INPUT statement calls for a string variable immediately following the word INPUT, then everything that is input for that line is put in the string variable. For example, if the input statement is

```
120 INPUT NAME$,AGE
```

and your response is

```
?SAM JONES,15
```

then the variable NAME$ will contain

```
SAM JONES, 15
```

and we will get another question mark to input the age. This illustrates a good rule of thumb:

RULE: **Don't mix numeric variables and string variables in the same INPUT statement!**

Try some short programs of your own until you are sure that you can use the INPUT statement correctly for both numeric and string variables. In the next section we'll look more closely at prompts.

Prompts

Every attempt should be made to make prompts in a program easy to read and understandable to the user. Type in and run this program as an example of what to *avoid*:

```
100 REM - Chapter 5, No. 9
110 REM - Sloppy Prompts
120 GRAPHICS 0:PRINT
130 PRINT "Type in an even number between 0
    and 100."
140 INPUT NUM
150 PRINT:PRINT "Your number is ";NUM
```

Poor, right? Change line 130 and add line 135 as follows:

```
130 PRINT "Type in an even number"
135 PRINT "between 0 and 100."
```

Do you see the difference? Always make the extra effort to make your prompts and other display text visually attractive. The following program shows how text can be balanced to look good on the screen. It is easy to use cursor control to insert or remove spaces between words to make text lines either longer or shorter. Note that line 160 holds the instructions on the screen before the actual prompts are printed.

```
100 REM - Chapter 5, No. 10
110 REM - Draw a line
120 GRAPHICS 7:POKE 752,1
130 PRINT "  Type in the  COORDINATES  of the"
140 PRINT "  ENDPOINTS of a LINE SEGMENT, and"
150 PRINT "  the line segment will be drawn."
160 FOR HOLD=1 TO 3000:NEXT HOLD
170 GRAPHICS 7:POKE 752,1
180 PRINT "First endpoint  (Column,Row) ";:INPUT X,Y
190 PRINT "Second endpoint (Column,Row) ";:INPUT A,B
200 COLOR 1
210 PLOT X,Y:DRAWTO A,B
220 PRINT:PRINT
230 PRINT "  Endpoints:  (";X;",";Y;") and (";A;",";B;")"
240 PRINT:PRINT "  Press SYSTEM RESET, then type"
250 PRINT "  RUN if you want to try again.";
260 GOTO 260
```

Experiment a bit by removing lines 170 and 220. Do you see that they are necessary to place the text on the correct line in the text window? We could also replace line 170 with PRINT statements. Can you figure out how many PRINTs would be necessary to make the prompts appear on the top line of the text window?

EXERCISES

1. An instruction to the user of a program that precedes an INPUT

    statement is called a _____ .

2. [( ) True ( ) False] A program that contains INPUT state-
    ments is called an interactive program.

3. If a program contains the statement INPUT AGE, and the
    user types in "JONATHAN", then an error message number

    _____ will result.

4. Fill the blanks in the runs for each of the following programs:

```
100 PRINT "Type three numbers, such as 5,3,7"
110 INPUT A,B,C
120 PRINT A
130 PRINT B,C

RUN
Type three numbers, such as 5,3,7
?-2,7,98
```

    _____

    _____


```
100 PRINT "Type 4 numbers, such as 8,11,-3,2"
110 INPUT U,V,W,X
120 PRINT U,U+V
130 PRINT W,W*X

RUN
Type 4 numbers, such as 8,11,-3,2

?_____,_____,_____,_____
7          12
3          45
```

5. The keyword INPUT has essentially the same function as
( ) PRINT   ( ) LET   ( ) LIST

6. [( ) True ( ) False] The statement 110 INPUT DATE$
will accept only a string as a user response.

7. Here is the printout for a program. You write the program.

```
RUN

Type the number of coins that you have, as follows:
QUARTERS?3
DIMES    ?5
NICKELS ?2

You have $1.35.

READY
```


8. Write an original program in which the user types in his present age,
and the computer responds by printing the year of his birth.
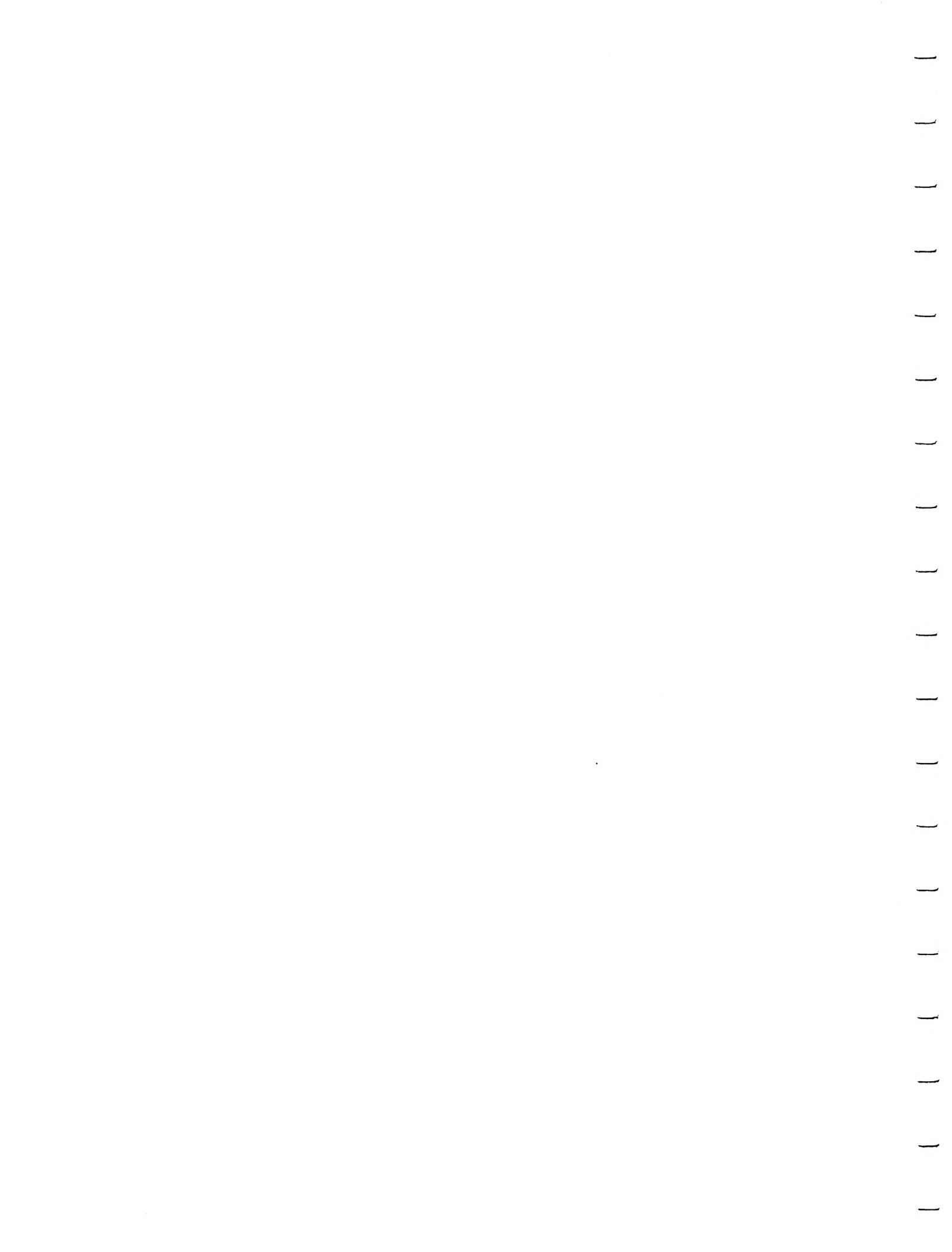
READ, DATA, AND RESTORE Statements

We have learned two ways of assigning values to variables:　　**Introduction**

1. LET statements

```
120 LET A=5
140 B=3.14159    ("LET" is optional.)
170 COURSE$="ENGLISH"
```

2. INPUT statements

```
130 INPUT A,B,C
160 INPUT DATE$
```

A third way to assign values to variables is to use READ and DATA statements.

The three following short programs use READ and DATA. Each　　**READ and DATA**
one will produce an ERROR 6- message; don't be alarmed!

```
100 REM - Chapter 6, No. 1
110 GRAPHICS 0:PRINT
120 PRINT "ADD TWO NUMBERS"
130 PRINT:PRINT
140 READ A
150 READ B
160 PRINT A;" + ";B;" = ";A+B
170 PRINT
```

```
180 GOTO 140
190 DATA 5,10,2,7,11,19,22,51
200 DATA 11.2,9.7,6.8,2.83

100 REM - Chapter 6, No. 2
110 GRAPHICS 3
120 COLOR 1
130 PLOT 11,1
140 READ X,Y
150 DRAWTO X,Y
160 GOTO 140
170 DATA 3,9,11,17,19,9,27,17,35,9,27,1,
    19,9,11,1

100 REM - Chapter 6, No. 3
110 DIM NAME$(10)
120 GRAPHICS 0:PRINT
130 PRINT "NAME","AGE"
140 PRINT "----","---"
150 READ NAME$,AGE
160 PRINT NAME$,AGE
170 GOTO 150
180 DATA BILL,15,LISA,14,JEFF,16,DEBI,15
```

Now let's see exactly how READ and DATA work. (We hope you still have Program Chapter 6, No. 3 on the computer.) You have probably deduced from the three preceding programs that the READ statement works almost exactly like the INPUT statement; but that instead of having the user input information (data), the data is included *in the program*. The statements that contain the data are known as (you guessed it!) DATA statements. Look now at lines 150 to 180 of the last program. These are the lines that READ and PRINT the data.

```
150 READ NAME$,AGE
160 PRINT NAME$,AGE
170 GOTO 150
180 DATA BILL,15,LISA,14,JEFF,16,DEBI,15
```

The data is read from line 180 in order from left to right, so that first NAME$ contains the name BILL, and AGE contains 15. These values are printed at line 160, then line 170 tells the computer to go back and execute line 150 again. The first two data items (BILL,15) are already used, so that the second time, LISA is read into NAME$, and 14 is read into AGE. The third time around, JEFF and 16 are read; and the fourth time, DEBI and 15. But now the computer tries to execute line 150 a fifth time. All of the data has been read from line 180, so we get the error message

ERROR-   6 AT LINE 150

which means "out of data." Obviously the computer can't read information that isn't there! It is very important that you understand that the *data is used up as it is read.*

Another important point is that the items in the DATA statement(s) must correspond in *type* with the variables in the READ statement. In the last program, line 150 calls for the computer to read first a string variable, and then a numeric variable; so the items in line 180 are arranged in the same order.

DATA statements may be placed anywhere in the program (like REM statements, they are not executed), but a common programming practice is to put them out of the way at the end of the program.

Certainly it is inelegant to end a program with an error message, so we need a way to have the computer stop normally after it has read all of the data. We already know how to have the computer "count" to a certain number to hold a musical note or to keep a graphics display on the screen. We can use the same technique to count how many times the READ statement is executed. Add line 145 to the last program, and change line 170 as follows:

```
145 FOR COUNT=1 TO 4
170 NEXT COUNT
```

Now exactly four sets of data are processed, and then the program will stop normally. Note that the NEXT statement replaces the GOTO statement. Can you make appropriate additions and corrections to the first two programs at the beginning of this chapter so that the programs stop normally?

Try two more examples of programs with READ and DATA statements:

```
100 REM - Chapter 6, No. 4
110 DIM NAME$(10)
120 GRAPHICS 0:PRINT
130 PRINT "Average of Two Grades":PRINT:PRINT
140 PRINT "NAME    "," 1    2    AVERAGE"
150 PRINT "--------","--    --    -------"
160 FOR COUNT=1 TO 4
170 READ NAME$,G1,G2
180 AVE=(G1+G2)/2
190 PRINT NAME$,G1;"    ";G2;"    ";AVE
200 NEXT COUNT
210 DATA Louise,85,91
```

```
220 DATA Roger,79,87
230 DATA DeeDee,80,71
240 DATA Michael,92,87
```

Pay close attention to the spacing and punctuation in lines 140, 150, and 190 in the program above. Notice that the average of the two grades is calculated at line 180. Here is an example of a music program using READ and DATA statements:

```
100 REM - chapter 6, No. 5
110 GRAPHICS 2:POSITION 2,4:PRINT #6;"NAME THIS TUNE!"
115 FOR COUNT=1 TO 31
120 READ PITCH,DURATION,SLUR
129 REM *** PLAY NOTE ***
130 SOUND 0,PITCH,10,10
139 REM *** HOLD NOTE FOR PROPER DURATION ***
140 FOR HOLD=1 TO 25*DURATION:NEXT HOLD
149 REM *** TURN NOTE OFF ***
150 SOUND 0,0,0,0
159 REM *** KEEP NOTE OFF FOR PROPER INTERVAL ***
160 FOR OFF=1 TO 5*SLUR:NEXT OFF
170 NEXT COUNT
180 DATA 35,2,10,35,2,10,44,2,10,44,2,10
190 DATA 35,2,10,35,1,1,35,1,10,29,4,10
200 DATA 40,2,10,40,2,10,47,2,10,47,2,10
210 DATA 40,2,10,40,1,1,40,1,10,33,4,10
220 DATA 35,2,10,35,2,10,44,2,10,44,2,10
230 DATA 35,2,10,35,1,1,35,1,10,29,4,10
240 DATA 40,2,10,33,1,1,33,1,10,35,2,10,40,2,10
250 DATA 44,4,10,44,2,10
```

The last program probably needs some explanation. To play a simple tune, we need to consider three things for each note:

1. **The pitch (middle C, A, B*b*, D#, etc.)**
2. **The duration (eighth, fourth, or half note, etc.)**
3. **The slur (is the note tied to the next one?)**

We have a table of pitch values in Appendix 2. These are the values in the DATA statements such as 35, 44, 29, 40, etc. In our tune, we have just eighth, fourth, and half notes. If the duration of an eighth note is 1, then the duration of a fourth note is 2 (twice as long), and the duration of a half note is 4 (four times as long). These numbers are the ones following the pitch values. Finally, if a note is tied to the next one (a slur), then the interval between the notes (when the first note is turned off) must be much shorter than normal. Thus, we have a slur value of 10 for a normal interval, and a slur value of 1 for a slurred interval. Whew! Don't worry right now if you find this difficult—we'll come back to it later. One last comment, though: You can "play" the song faster or slower by decreasing or increasing the number 25 in line 140. This is the multiplier for the duration.

We have emphasized that data gets used up as it is read. Sometimes, though, it would be useful to use the same data (or part of the data) again. The RESTORE statement makes this possible. First, type in and run this program to play a few simple notes:

```
100 REM - chapter 6, No. 6
110 FOR COUNT=1 TO 7
120 READ PITCH
130 SOUND 0,PITCH,10,10
140 FOR HOLD=1 TO 100:NEXT HOLD
150 NEXT COUNT
200 DATA 121,96,81,60,81,96,121
```

The program "plays" seven notes and then stops normally. If we want to play the notes a second time (without typing RUN again), we can't go back to line 110 because the data is used up. We need to RESTORE it. Add these lines to your program:

```
160 SOUND 0,0,0,0
170 FOR OFF=1 TO 200:NEXT OFF
180 RESTORE
190 GOTO 110
```

Now, after seven notes are played, the sound is turned off for a count of 200, the data is RESTORED (made usable again), and the notes are played again (and again, and again, ...).

Keep in mind that the RESTORE statement makes the data usable again *beginning with the first data item in the program.* This is important to remember because later you will work with programs that may have different sets of data for different parts of the program. Thus, for example, we may want to RESTORE only the data beginning on line 350. We simply add the line number after the RESTORE, as:

```
180 RESTORE 350
```

We use this same statement in the following program, which draws a square pattern of dots and then makes the inside of the square "disappear" (the points are replotted in black, the background color). Note that the data is listed so that the outline of the square is drawn first, then the inside. This allows us to use the same "inside" data to replot the dots in black.

```
100 REM - Chapter 6, No. 7
110 GRAPHICS 3
```

```
120 COLOR 1
130 FOR COUNT=1 TO 25
140 READ X,Y
150 PLOT X,Y
160 NEXT COUNT
170 FOR WAIT=1 TO 300:NEXT WAIT
180 RESTORE 350
190 COLOR 0
200 FOR COUNT=1 TO 9
210 READ X,Y
220 PLOT X,Y
230 NEXT COUNT
299 REM *** DATA FOR OUTLINE OF SQUARE ***
300 DATA 16,6,18,6,20,6,22,6,24,6
310 DATA 16,8,24,8
320 DATA 16,10,24,10
330 DATA 16,12,24,12
340 DATA 16,14,18,14,20,14,22,14,24,14
350 DATA 18,8,20,8,22,8
360 DATA 18,10,20,10,22,10
370 DATA 18,12,20,12,22,12
```

If you are satisfied that you understand the function of line 180, then add the following lines to produce a "blink."

```
240 RESTORE
250 GOTO 120
```

Note that line 240 RESTOREs the data beginning with line 300 (the first data), so that the display can begin again.

In the next section we'll look at a way to save some keystrokes in typing in a program.

Abbreviations

Most of the keywords (commands) that we have used so far may be abbreviated when they are typed. When the program is LISTed (either on the screen or printer), then the full keyword is printed. Here is a list of the keywords we have studied and their abbreviations. An asterisk (*) at the left of a keyword indicates that the abbreviation is useful. (The others don't save any keystrokes.)

Try this program, which is almost a one-liner:

```
100 . - Chapter 6, No. 8
110 GR.3:SE.0,4,4:C.1:F.K=1 TO 18:PL.1,
    K:DR.38,K:N.K
120 PR."ABBREVIATION PRACTICE"
```

TABLE 6-1 Keyword Abbreviations

Keyword	Abbreviation
CLOAD	CLOA.
CSAVE	none
* COLOR	C.
* DATA	D.
DIM	DI.
DOS	DO.
* DRAWTO	DR.
* FOR	F.
* GOTO	G.
* GOSUB	GOS.
* GRAPHICS	GR.
IF	none
* INPUT	I.
LET	LE. ("LET" is optional.)
* LIST	L.
* LOAD	LO.
* LPRINT	LP.
* NEXT	N.
* PLOT	PL.
POKE	POK.
* POSITION	POS.
* PRINT	PR. or ?
READ	REA.
* REM	. (preceded by a space)
* RETURN	RET.
* RESTORE	RES.
RUN	RU.
* SAVE	S.
* SETCOLOR	SE.
* SOUND	SO.
STOP	STO.
THEN	none

EXERCISES

1. The READ statement is similar in function to the _____ and _____ statements.

2. [( ) True   ( ) False]   If the statement 200 DATA 5,10,15 is executed, the numbers 5, 10, and 15 will be printed on three consecutive lines.

3. If a program containing a READ statement "runs out of data," then

    an error message message number _____ will result.

4. In each of the four short programs below, give the *last* value
    assigned to the variable A.

    ```
    100 READ A,B,C
    110 DATA -12,10,5
    ```

    ```
    100 READ A,A1,A2
    110 DATA 5,-6,-9,34,23
    120 READ X,A
    ```

    ```
    100 LET A=0
    110 READ X1,Z1,A1
    120 LET A=A+A1
    130 LET A=A*(A1-Z1)
    140 DATA 12,14,26
    ```

    ```
    100 DATA 1,3,5,7,9
    110 READ X,Y,A
    120 DATA 2,4,6
    ```


5. Complete the READ statement to read the data in the statement

    ```
    250 DATA SUNDAY,15,8.
    ```

    ```
    130 READ _____
    ```


6. Check all of the following statements that are legitimate.

    ```
    ( )    315 DATA 12%,15%,17%
    ( )    242 READ 23.5,61.2
    ( )    290 DATA "GOSH!","JEEPERS!","GEE WHIZ"
    ( )    240 READ NAME$,ADDRESS$
    ( )    510 DATA JOE,PETE,LISA,PAULA,GEORGE,TINA
    ( )    285 DATA D-A-S-H,S P A C E
    ```


7. Rewrite (in the space at the right) each of the following programs
    so that it will run correctly.

    ```
    100 READ X;Y
    110 LET P=XY
    120 PRINT "PRODUCT = ";P
    130 DATA 6,8
    ```

```
100 LET R=S
110 READ S
120 LET X=R+2*S
130 PRINT "X = "X
```

```
100 READ S
110 LET T=S*S
120 PRINT SQUARE IS T
130 DATA 6,8
```

```
100 READ X,Y,Z
110 LET A=(X+Y+Z)/3
120 PRINT "AVERAGE IS ";A
130 DATA 821.4     287.9
```

8. Rewrite (in the space at the right) each of the following programs so that it will run correctly.

```
100 READ M,N
110 LET Q=MN+3
120 PRINT Q
130 DATA 78,91
```

```
100 DATA 21
110 R E A D R
120 LET 4*R=S
130 PRINT S
```

```
100 READ (C,D)
110 LET X=C+D
120 PRINT "SUM IS X"
130 DATA 37,21
```

```
100 READ X+Y,Z
110 LET A=X+Y/Z.
120 PRINT "RESULT IS"R
130 DATA 84.1,34,26
```

_____
_____
_____
_____

9. [( )True ( )False]   Once the data in a program has been "used up," there is no way to use it again.

10. Give the full BASIC keyword for each of the following abbreviations:

? _____   GR. _____   LO. _____

I. _____   PL. _____   SE. _____

RES. _____   SO. _____   L. _____

11. Write the printout of the following program:

```
100 LN=200
110 READ A,B,C
120 PRINT (A+B)/C
130 LN=LN+10:RESTORE LN
140 GOTO 110
200 DATA 8
210 DATA 4
220 DATA 6
230 DATA 5
240 DATA 11
```

12. Write an original program that will add up 10 numbers and print their average. Use a READ statement to read the numbers from a DATA statement. Use the following numbers: 19,22,15.3,16.1,27,23,9.7,12,20.56,15.8.

13. Write an original program similar to the one in **12.** but which prints the average of the _first five_ numbers and then the average of the _last five_ numbers of a set of eight numbers. Use a RESTORE statement and the following data: 5,11,17,4,3,8,9,12. (Hint: Put the data in two DATA statements.)

Transfer of Control Statements

We know that the computer normally executes program statements in numerical order according to their line numbers. In this chapter we are going to consider ways to change this order of execution.

We have already used the GOTO statement extensively, particularly in Chapter 6 with READ and DATA statements. Try this simple program as a review:

```
100 REM - Chapter 7, No. 1
110 GRAPHICS 3
120 COLOR 1
130 PLOT X,Y
140 SOUND 0,4*(X+Y),10,10
150 FOR WAIT=1 TO 50:NEXT WAIT
160 X=X+2:Y=Y+1
170 GOTO 130
```

In this example, control of the program is transferred at line 170 back to line 130. This is known as an *unconditional branch* (transfer of control) because there is only one option (control must always transfer to line 130). Change line 170 as follows, and see what happens:

```
170 GOTO 125
```

83

You should get this error message:

```
ERROR-    12 AT LINE 170
```

This means "line not found." There is no line 125 in the program. Be sure that your GOTO statements give the line number of a line that is actually in your program.

Loops

In the preceding program, lines 130 to 170 were repeated over and over until we received a "cursor out of range" error (141). A sequence of statements in a program that is executed repeatedly is called a *loop*. Our next concern is to find ways to terminate a loop. (The preceding program was terminated by an error message, which is certainly not very elegant.) The next program contains an *endless loop*:

```
100 REM - Chapter 7, No. 2
110 GRAPHICS 0:PRINT
120 PRINT "Type a number ";:INPUT NUM
130 PRINT NUM;" squared is ";NUM*NUM
140 PRINT
150 GOTO 120
```

Do you see why we said the program contains an endless loop? It just keeps going on and on. One way to stop an endless loop is to press the BREAK key. You will get a message such as

```
STOPPED   AT LINE 130
```

where 130 is the number of the line that was being executed as you pressed BREAK. This method of terminating a loop is useful on certain occasions, but like the previous example with an error message, it has a big disadvantage: *No other part of the program can be executed without additional action on the part of the user.* We need a method of terminating loops that depends somehow on certain conditions in the program. Such a method is provided by the IF...THEN statement.

The IF...THEN Statement

Change line 120 and add line 125 to the preceding program as follows:

```
120 PRINT "Your number (type -1 to stop) ";:INPUT NUM
125 IF NUM=-1 THEN GRAPHICS 0:STOP
```

Now, when you run the program, you can make it terminate by giving "-1" as a response to the prompt. We call line 125 a *conditional branch* because control is transferred only if a certain condition is met; namely, we type in -1 as a response.

Let's consider the structure of the IF...THEN statement. Following the IF, we can place any expression that could be classified as true or false. Following the THEN, we can place either (1) a line number, or (2) a command (or several commands separated by colons) that can be executed by the computer. Here are some examples:

```
120 IF X=5 THEN 300
150 IF NAME$="ZZZ" THEN PRINT "FINISHED":GOTO 500
190 IF AREA>1000 THEN STOP
210 IF X<Y THEN X=Y:GOTO 150
170 IF B*B-4*A*C<0 THEN PRINT "NO REAL ROOTS":GOTO 110
310 IF R$="YES" THEN LET A$="TOTAL"
```

In most cases where we use the IF...THEN statement, the expression following the IF usually compares two quantities or two strings. Here is a chart of the symbols to use for comparisons. Be careful to write the double symbols in the proper order.

TABLE 7-1	Comparison Symbols	
BASIC	*English*	*Example*
=	equal to	5=5
<	less than	2<3
>	greater than	10>7
<>	not equal to	2<>9
<=	less than or equal to	2<=3
>=	greater than or equal to	8>=6

Here is a program that uses IF...THEN statements to test for three different possibilities. Please note that lines 160, 170, and 180 occupy two lines on the screen. They are shown below exactly as they will appear on the screen as you type them in. For each of these three program lines, don't press the RETURN key until you have finished the second screen line.

```
100 REM - Chapter 7, No. 3
110 GRAPHICS 0:PRINT
120 PRINT "How old will you be in"
130 PRINT "November of this year ";
140 INPUT AGE
150 PRINT:PRINT
160 IF AGE=18 THEN PRINT "This is your
    first voting year.":GOTO 200
```

```
170 IF AGE>18 THEN PRINT "You have vot
ed for ";AGE-18;" years.":GOTO 200
180 PRINT "You may vote in ";18-AGE;"
more years."
200 REM *** END OF PROGRAM ***
```

Be sure to try all three possibilities for an input: 18, a number less than 18, and a number greater than 18.

Pay close attention to the way the IF...THEN statement works:

1. **If the expression following IF is *true*, then the statement(s) following THEN are executed. These statements may or may not include a branching (GOTO) statement. If they do not, then control passes to the next statement in the program.**

2. **If the expression following IF is *false*, then the statement(s) following THEN are ignored and control passes to the next statement in the program.**

Do you see why we didn't need a line in our program that started "IF AGE<18 THEN..."? Right! If AGE is not *equal* to 18 (line 160), and AGE is not *greater than* 18 (line 170), then it MUST be *less than* 18 (line 180).

See what will happen if you run the program and input an impossible age, such as -10. (There are always wise guys around.) Because the input is meaningless, the computer's response is also meaningless. We need a test to catch inappropriate inputs.

We can make such a test by adding just one line to our program. Note that it occupies three lines on the screen—don't press the RETURN key until you have typed all three lines.

```
155 IF AGE<0 THEN PRINT "Oops!  Your a
ge must be positive!":FOR WAIT=1 TO 10
00:NEXT WAIT
```

Now, if you respond with a negative age, you will get a gentle reprimand and another chance. We always need to anticipate inappropriate inputs on the part of the user. To err is human!

Do you think the following IF...THEN statement makes sense?

```
170 IF A$<B$ THEN PRINT A$;" comes before ";B$
```

Can we compare strings as if they were numbers? The answer is *yes*, as the following program illustrates:

```
100 REM - Chapter 7, No. 4
110 DIM A$(10),B$(10)
120 GRAPHICS 0:PRINT
130 PRINT "Type in two names:"
140 PRINT "1) ";:INPUT A$
150 PRINT "2) ";:INPUT B$
160 PRINT:PRINT
170 IF A$<B$ THEN PRINT A$;" comes before ";B$:GOTO 200
180 IF A$>B$ THEN PRINT B$;" comes before ";A$:GOTO 200
190 PRINT A$;" equals ";B$
200 REM *** END OF PROGRAM ***
```

So! The expression "A$<B$" is

1. **_True_ if the strings stored in A$ and B$ are in the alphabetical order A$,B$.**

2. **_False_ if the strings stored in A$ and B$ are either equal, _or_ in the alphabetical order B$,A$.**

This capability of comparing strings is the basis of sorting lists of words or names into alphabetical order, as we shall see later.

Lengths of Lines

If a comparison is _true_ in an IF...THEN statement, then often we want to do several things, as in the voting program Chapter 7, No. 3. How long may a line be?

One line across the screen of the TV set or monitor is known as a _physical line_; normally it is 38 characters long. You already know that some of our program statements have extended onto the second (and even third) physical line. Each numbered statement in a program is known as a _logical line_ and may consist of a maximum of three physical lines (3 × 38, or 114 characters). You are probably already familiar with the warning buzzer that sounds as you approach the end of the third physical line in a program statement. If you exceed three physical lines in a logical line, the extra characters will be dropped. In this case, we say that the line has been _truncated_. If a logical line of more than 138 characters does creep into your program anyway, an error message 14 (line too long) will result.

The STOP and END Statements

Try this program to see the difference between STOP and END. On the second run, change STOP in line 300 to END.

```
100 REM - Chapter 7, No. 5
110 GRAPHICS 0:PRINT
120 PRINT "CUBES LESS THAN 500"
130 PRINT "-------------------"
```

```
140 PRINT
150 PRINT "   NUMBER        CUBE"
160 PRINT "   ------        ----"
170 NUM=1
180 CUBE=NUM*NUM*NUM
190 IF CUBE>500 THEN 300
200 PRINT "      ";NUM;"              ";CUBE
210 NUM=NUM+1
220 GOTO 180
300 STOP
```

You should have discovered that after the display the following messages are printed, depending on line 300:

```
300 STOP                          300 END
------------------                -------
STOPPED   AT LINE 300             READY
                                  ■
```

In either case, we are now back in direct (immediate) mode, and the computer is waiting for further instructions from us. There is an important difference between STOP and END, though: The STOP statement will not turn off sounds, but the END statement will. In either case, you may continue the execution of the program from the instruction immediately following END or STOP by typing CONT (continue) and pressing the RETURN key. This gives us one of several techniques for correcting *bugs* (errors) in programs.

Debugging (Error Correction)— Part 1

The error correction technique just mentioned involves three steps:

1. Divide the program into sections

2. Insert a STOP statement after each section

3. Correct any errors found in that section

The following program prints "HI!" in GRAPHICS 3 block letters. There is a STOP statement after each of the sections that prints a letter so that you can check letter by letter to see if your data is correctly typed. As soon as a section ("H", "I", "!") works all right, then remove the STOP statement after it. To continue after a STOP, type CONT and press the RETURN key.

```
100 REM - Chapter 7, No. 6
110 DIM A$(1)
120 GRAPHICS 3:COLOR 1
```

```
124 REM *** PRINT "H" ***
125 FOR K=1 TO 6
130 READ A$,X,Y
135 IF A$="P" THEN PLOT X,Y:GOTO 145
140 IF A$="D" THEN DRAWTO X,Y
145 NEXT K
146 STOP
149 REM *** PRINT "I" ***
150 FOR K=1 TO 6
155 READ A$,X,Y
160 IF A$="P" THEN PLOT X,Y:GOTO 170
165 IF A$="D" THEN DRAWTO X,Y
170 NEXT K
171 STOP
174 REM *** PRINT "!" ***
175 FOR K=1 TO 3
180 READ A$,X,Y
185 IF A$="P" THEN PLOT X,Y:GOTO 195
190 IF A$="D" THEN DRAWTO X,Y
195 NEXT K
196 STOP
500 REM *** H ***
510 DATA P,13,3,D,13,15,P,18,3,D,18,15,P,14,9,D,17,9
520 REM *** I ***
530 DATA P,22,3,D,22,15,P,21,3,D,23,3,P,21,15,D,23,15
540 REM *** ! ***
550 DATA P,27,3,D,27,13,P,27,15
```

At each STOP you may type PRINT X,Y and press RETURN to see what the least values of X and Y are. If they do not correspond to the DATA statement above, then make changes in that line of your program and try again. If you want a list of just part of your program, for example, the part that prints "I", type LIST 149,171 and press RETURN. Remember, the point here is to use STOP to look at your program section by section. (Do you see how the P and D codes work in the plotting routine? More about it later.)

EXERCISES

1. A sequence of statements in a program that is executed more than

    one time is called a _____ .

2. [( ) True   ( ) False]   The statement 200 GOTO 130 will produce a conditional branch to line 130.

3. If a program contains the statement 190 GOTO 570 but there is no

    line 570 in the program, then error message number _____ will
    result.

4. Write each of the following comparisons as a BASIC statement.
    Variable names are capitalized. The first one is done for you as an
    example.

    If A is equal to B   IF A=B

    If AGE is less than WEIGHT   _____

    If HIGH is greater than LOW   _____

    If X does not equal Y   _____

    If G is less than or equal to K   _____

    If FUEL is not less than ROCKETS   _____

5. The normal length of a physical line on the screen is

    _____ characters.

6. When either a STOP or END statement is executed, the

    computer returns to _____ mode.

7. In each of the three programs below, a required GOTO
    statement has been omitted. The GOTO statement should
    cause a branch to process another set of data. Supply the
    needed statement.

```
100 DATA 10,20,5,15,-6,14
110 READ A,N
120 PRINT "A + N = ";A+N

130 _____



100 INPUT X,Y,Z
110 PRINT "X + Y x (-Z) = ";X+Y*(-Z)

120 _____
```

```
100 LET A=5
110 LET B=6
120 PRINT A+B,A*B
130 LET A=A+1
140 LET B=B+2

150 _____
```

8. Rewrite (in the space at the right) each of the following two programs so that each will run correctly.

```
100 READ X,Y                    _____
110 DATA 73,44                  _____
120 GOTO 140                    _____
130 READ T,U                    _____
140 DATA 12,-21                 _____
150 PRINT X,Y,T,U               _____
```

```
100 PRINT "NAME","HOURS"        _____
110 READ NAME$,HOURS            _____
120 DATA WARD,30,GRAY,55        _____
130 PRINT NAME$,HOURS           _____
140 GOTO READ                   _____
```

9. Write the printout of each of the following two programs *without* using the computer.

```
100 READ J,K           ¦
110 LET S=J+K-J*K      ¦
120 GOTO 100           ¦
130 PRINT S            ¦
140 DATA 2,3,4,5,6,8   ¦
```

```
100 READ X,Y             ¦
110 LET Z=X-3*Y          ¦
120 PRINT "ANSWER IS ";Z ¦
130 DATA 2,-3,4          ¦
140 DATA 0,5,-1          ¦
```

10. After the program LISTed below, you are given different possibilities for line 130. In each case, state if the condition is true or false, why, and what line will be executed next. The first one is done for you as an example.

```
100 LET A=5
110 LET B=3
120 LET C=-4
130 IF...
140 ...
...
...
```

```
130 IF A>C THEN 190
```

```
130 IF C>A THEN 280
```

```
130 IF B<=C THEN 210
```

```
130 IF B>A*C THEN 185
```

```
130 IF A+C>B THEN 200
```

```
130 IF A>=B-C THEN 175
```

11. [( )True   ( )False]   An END statement may occur anywhere in a program.

12. What command should be typed in to have the execution of a program resume at the line after a STOP statement? _____

13. In debugging a program, where should STOP statements be in serted? _____

14. Write an original program using one or more IF...THEN statements and an INPUT statement that will tell if a user's number is a perfect square. Some sample *runs* are shown.

```
RUN

Type a positive number.
(Type -1 to stop.)
?43

43 is not a square.

READY
■
```

RUN

Type a positive number.
(Type -1 to stop.)
?196

196 is 14 squared.

READY
█

RUN

Type a positive number.
(Type -1 to stop.)
?-1
Good-by.

READY
█

8
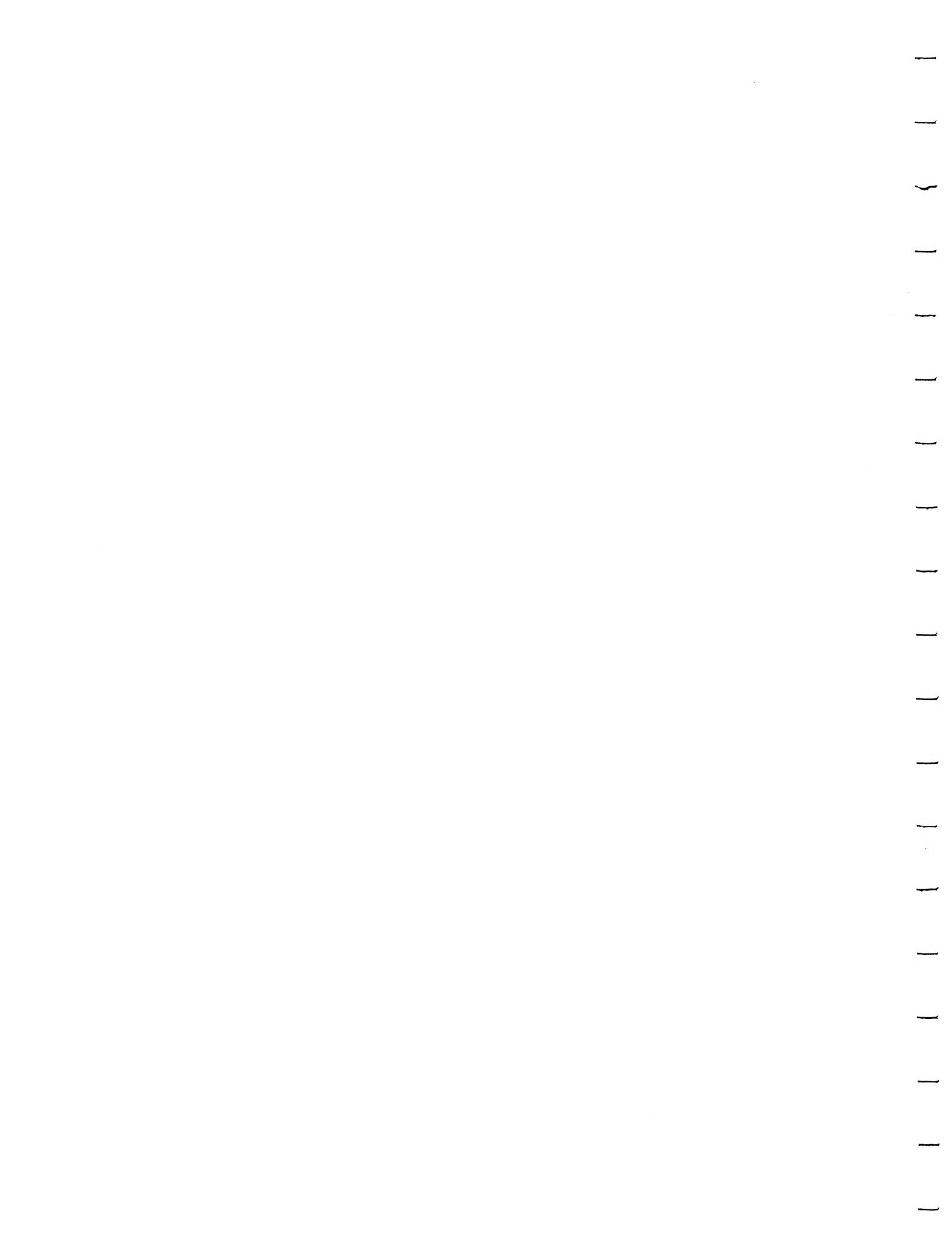
Anatomy of a Loop

In the last chapter, we introduced the concept of a loop. In this chapter, we are going to look more closely at the various parts of a loop. The following program will serve as the reference program for the next three sections, so type it in very carefully. Pay particular attention to the spacing in lines 150, 160, and 180. (Use lines 120 and 130 as a guide, so that the proper columns line up in all lines.)

```
100 REM - Chapter 8, NO. 1
110 GRAPHICS 0:PRINT
120 PRINT "SQUARES OF NUMBERS 10-20"
130 PRINT "-----------------------"
140 PRINT
150 PRINT "    NUMBER      SQUARE"
160 PRINT "    ------      ------"
170 NUM=10
180 PRINT "        ";NUM;"           ";NUM*NUM
190 IF NUM=20 THEN END
200 NUM=NUM+1
210 GOTO 180
```

Lines 170 to 210 are the real "heart" of this program; they are the lines that do the calculations and that print the results for the numbers from 10 to 20. These lines form the loop. Every loop necessarily contains one (or more than one) variable that changes as the loop is repeated. In this program, that variable is NUM. In line 170, the initial value of NUM is set to 10; that is,

NUM will vary, but its first (initial) value is 10. The process of giving starting values to variables is called *initialization*. Every variable in a program will have to have some initial value. Often that value is zero, and because the ATARI automatically sets variables equal to zero (remember?), we never need to initialize a variable to zero. Try this quick experiment: change line 170 to

```
170 REM *** NO INITIALIZATION ***
```

and run the program. What happens? You get a display of numbers from 0 to 20 and their squares, with the titles scrolled off the screen. Try changing line 170 as indicated below and then run the program again.

```
170 NUM=15
```

What happened? You got the numbers from 15 to 20 and their squares, and this time the titles didn't scroll off the screen. However, the title isn't correct anymore; we need to change line 120 for a "perfect" program. Now try this change for line 170:

```
170 NUM=-5
```

Well, now we know it works for negative numbers! One more experiment before we move on: change line 170 to

```
170 NUM=25
```

Help! Try the BREAK key. Do you see what is wrong? NUM starts at 25 and *increases* by one each time through the loop, but line 190 will stop the program ONLY if NUM is exactly equal to 20. Thus, we have an endless loop. It should be clear that endless loops are wasteful and, on large computers, very expensive. Beware!

Let's review intialization quickly:

1. A variable in a program may be set to an initial value with a LET statement.
2. Variables that are not initialized in this way are automatically initialized to zero.

Don't NEW your program; we're going to look at other features of it.

In line 200 of the program, the current value of NUM is increased, or incremented, by one. The value one (1) is called the *increment*, and the whole process is called *incrementation*. Without an incrementation, we would be in an endless loop, and the display would show the same pairs of numbers over and over. (If you're curious, just delete line 200 and then run the program.) Let's try a different increment: change line 200 to

```
200 NUM=NUM+2
```

Did you get the even numbers from 10 to 20? Suppose you make the increment 3:

```
200 NUM=NUM+3
```

Oops! We went past the test! You can see that care is involved here. How about a negative increment (sometimes called a *decrement*)? Make the following two changes:

```
170 NUM=30
200 NUM=NUM-1
```

Good! We can make loops go forward or backward! Increments needn't be whole numbers; they may also be decimals. Try these changes:

```
170 NUM=5.1
190 IF NUM=5.9 THEN END
200 NUM=NUM+0.2
```

So, we see that increments may be positive or negative whole numbers or decimals. There is still one more possibility: The increment may also be a variable. Suppose we modify the program so that the user can determine the increment for each run of the program. We'll store the increment in the variable INCRNUM (the increment of NUM), and add a prompt for the user. Make these changes:

```
142 PRINT "What increment do you want ";:INPUT INCRNUM
144 PRINT:PRINT
200 NUM=NUM+INCRNUM
```

Now we can try whatever increment we like without having to change the program—an increase in the flexibility of the

program. However, some increments (as we discovered on the previous page) are inconsistent with the test at line 190 as we have it written. We need to increase the flexibility of the test as well. We'll refer to tests as *decisions*.

Decisions

The test, or *decision*, statement from our program is line 190:

```
190 IF NUM=20 THEN END
```

We can quickly solve some of our problems by changing this to

```
190 IF NUM>=20 THEN END
```

Now, the program will end if the last value printed is either

1. **Exactly equal to 20, or**
2. **Greater than 20**

Try running the program once with an increment of 2, and then again with an increment of 3 to see how this works.

We have already looked at the structure of a decision (IF...THEN) statement in Chapter 7, but decision statements are so important that we'll review how they work. Taking line 190 of our program as an example:

1. If the value of NUM is exactly equal to 20, the program ends.
2. If the value of NUM is greater than 20, the program ends.
3. If the value of NUM is less than 20, line 200 is executed next.

We can think of the command in line 190 as asking a question: Is the value of NUM greater than (or equal to) 20? If the answer is *yes*, then the program ends; if the answer is *no*, then NUM is incremented and we go through the loop again. Whenever we write a decision statement, we must be perfectly clear about two things:

1. What we want to do if the question asked in the statement has a *yes* answer
2. What we want to do if the question asked in the statement has a *no* answer

Let's ask the question in line 190 in a different way:

```
190 IF NUM<20 THEN ...
```

What should we write in place of the "..."? Well, if the value of
NUM is less than 20, then we need to increment NUM and then
go back to line 180. If the value of NUM is *not* less than 20, that
is, it is *greater than* 20, or *equal to* 20, then the program should
end. Then we could rewrite our loop as follows, eliminating
line 210:

```
180  PRINT "          ";NUM;"          ";NUM*NUM
190  IF NUM<20 THEN NUM=NUM+INCRNUM:GOTO 180
200  END
```

Does this seem easier, or harder, to understand than our
original loop (below)?

```
180  PRINT "          ";NUM;"          ";NUM*NUM
190  IF NUM>=20 THEN END
200  NUM=NUM+INCRNUM
210  GOTO 180
```

We might anlayze the structure of a loop as shown in Figure
8-1.
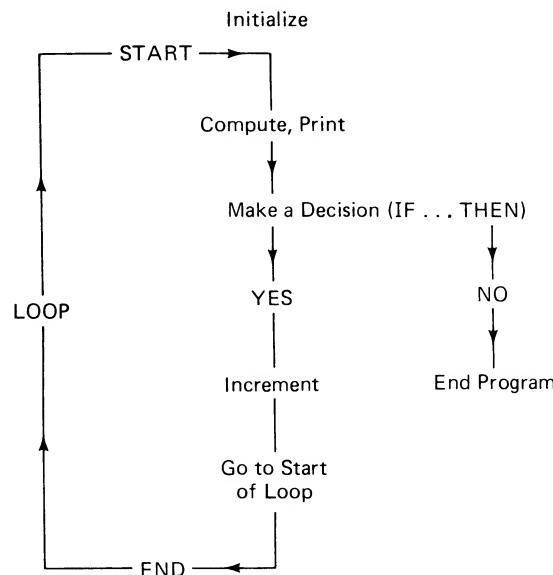
FIGURE 8-1   Structure of a Loop

Please note that Figure 8-1 refers to a loop in which the question
asked in the decision statement is: Should we stay in the loop?
This corresponds to the version of our program, which has line
190 as

```
190  IF NUM<20 THEN NUM=NUM+1:GOTO 180
```

Now that we have a good understanding of decisions, let's see if we can "fix" the first program in Chapter 7 ("Chapter 7, No. 1"). We'll repeat it here for convenience:

```
100 REM - Chapter 8, No. 2
101 REM - (Chapter 7, No. 1 repeated)
110 GRAPHICS 3
120 COLOR 1
130 PLOT X,Y
140 SOUND 0,4*(X+Y),10,10
150 FOR WAIT=1 TO 50:NEXT WAIT
160 X=X+2:Y=Y+1
170 GOTO 130
```

Do you remember that we always got an error message (141–cursor out of range) when we ran the program? Let's build a test into the program so that it will stop *before* it tries to plot a point with coordinates that are too large. We'll phrase our question according to the structure diagram we just discussed. Recall that in GRAPHICS 3 the columns are numbered from 0 to 39 and the rows are numbered from 0 to 19. Thus, 39 and 19 are the largest permissible values for X and Y in line 130. As long as X is less than 39, *and* Y is less than 19, we can stay in the loop. Add the following statements:

```
101 REM - (Addition to Chapter 8, No. 2)
160 IF X<38 AND Y<19 THEN X=X+2:Y=Y+1:GOTO 130
170 END
```

Did it work? The display should end by printing READY and the cursor in the text window. We used 38 (instead of 39) in line 160 because all of the values of X are *even* numbers because the first value is 0 and the increment is two. Forty (40), the next even number after 38, would be too large.

Our decision statement in line 160 has a new feature: The use of the keyword AND. The statement "X<38 and Y<19" is *true* only if *both* of the statements "X<38" and "Y<19" are *true*. We want to be sure that both the X and Y values are in range, so we use a *compound* (two or more parts) IF statement. Notice also that following the THEN statement we do three things: (1) increment X, (2) increment Y, and (3) go back to line 130.

You can think of line 160 as asking the question: "Are X and Y *small enough* to go through the loop again?"

In the preceding two programs (and their variations), the decision statement provided for two possibilities: Stay in the loop, or end the program. Actually, we needn't end the program;

we can do something else after the loop is finished. Let's add a loop to our preceding program that will draw the other dotted diagonal of the rectangle. It will be very similar to the existing loop. Add these lines:

```
100 REM - Chapter 8, No. 3
101 REM - (Addition to Chapter 8, No. 2)
170 X=38:Y=0
180 PLOT X,Y
190 SOUND 0,4*(38-X+Y),10,10
200 FOR WAIT=1 TO 50:NEXT WAIT
210 IF X>0 AND Y<19 THEN X=X-2:Y=Y+1:GOTO 180
220 END
```

Why do we need line 170? Right! We want to start one square to the left of the upper right corner of the screen, so the initial value of X is 38, and the initial value of Y is 0. (We have to initialize Y because it last had the value of 19 from the first loop.) Now look at the sound statement in line 190. We would like the pitch value to get larger (the tones get lower) as the dotted line is drawn from upper right to lower left. The value 38-X *increases* from 0 to 38 as X *decreases* from 38 to 0, so the expression 4*(38-X+Y) will *increase* as we go through the loop. At line 210, we are asking: Is X still *large enough*, AND is Y still *small enough*, to go through the loop again? If they are, then X gets a negative increment and Y gets a positive increment.

The more you experiment at this point by modifying the initialization, incrementation, and decision statements in this program, the more you'll learn about writing loop programs. Next we'll look at a technique for understanding what a program does without running it.

Tracing a Program

Type in and run the following program:

```
100 REM - Chapter 8, No. 4
110 GRAPHICS 0:PRINT
120 COUNT=10
130 PRINT "HAVE A NICE DAY!"
140 IF COUNT>11 THEN END
150 COUNT=COUNT+1
160 GOTO 130
```

A *trace* of a program is a chart that shows the following items as each line of the program is executed: The line number, the values of variables, the results of tests, and the output. A trace of the preceding program follows in Table 8-1.

TABLE 8-1 Trace of Program Chapter 8, No. 4

Step Number	Statement Number	Value of "Count"	Decision (Test)	Yes /No	Output
1	110				(clear screen)
2	120	10			
3	130				HAVE A NICE DAY!
4	140		COUNT>11?	NO	
5	150	11			
6	160				
7	130				HAVE A NICE DAY!
8	140		COUNT>11?	NO	
9	150	12			
10	160				
11	130				HAVE A NICE DAY!
12	140		COUNT>11?	YES	
13	END				

To write the trace, we simply read each line of the program and record the results of executing that line. Because REMark statements aren't executed, we ignore them. When the first value for the variable *count* is entered (step 2), it remains the value until a new one is entered (steps 5 and 9). Each time the decision statement is executed, we show the answer to the question asked. We draw a horizontal line to show the end of each pass through the loop. In our program, we see that the loop is executed three times, so the sentence "HAVE A NICE DAY!" is printed three times.

Following are two programs that are similar to the preceding one but with the elements of the loop placed in a different order. First, write a trace of each program, using a chart like Table 8-1. Then type the programs in and run them to see if your trace is correct.

```
100 REM - Chapter 8, No. 5
110 GRAPHICS 0:PRINT
120 COUNT=10
130 IF COUNT>11 THEN END
140 PRINT "HAVE A NICE DAY!"
150 COUNT=COUNT+1
160 GOTO 130


100 REM - Chapter 8, No. 6
110 GRAPHICS 0:PRINT
120 COUNT=10
130 COUNT=COUNT+1
140 IF COUNT>11 THEN END
150 PRINT "HAVE A NICE DAY!"
160 GOTO 130
```

Here is the trace for the program Chapter 8, No. 5:

TABLE 8-2   Trace of Program Chapter 8, No. 5

Step Number	Statement Number	Value of "Count"	Decision (Test)	Yes /No	Output
1	110				(clear screen)
2	120	10			
3	130		COUNT>11?	NO	
4	140				HAVE A NICE DAY!
5	150	11			
6	160				
7	130		COUNT>11?	NO	
8	140				HAVE A NICE DAY!
9	150	12			
10	160				
11	130		COUNT>11?	YES	
12	END				

Here is the trace for program Chapter 8, No. 6:

TABLE 8-3   Trace of Program Chapter 8, No. 6

Step Number	Statement Number	Value of "Count"	Decision (Test)	Yes /No	Output
1	110				(clear screen)
2	120	10			
3	130	11			
4	140		COUNT>11?	NO	
5	150				HAVE A NICE DAY!
6	160				
7	130	12			
8	140		COUNT>11?	YES	
9	END				

The important message of these three different traces is that the order of the elements in a loop (and the values in the decision statement) will determine how many times the loop is executed. Here is a "model" program to print "HAVE A NICE DAY!" five times:

```
100 REM - Chapter 8, No. 7
110 GRAPHICS 0:PRINT
120 COUNT=1
130 PRINT "HAVE A NICE DAY!"
140 IF COUNT=5 THEN END
150 COUNT=COUNT+1
160 GOTO 130
```

The structure of the loop in the preceding program could be diagrammed as shown in Figure 8-2 (compare with the diagram of a loop in the section on Decisions).
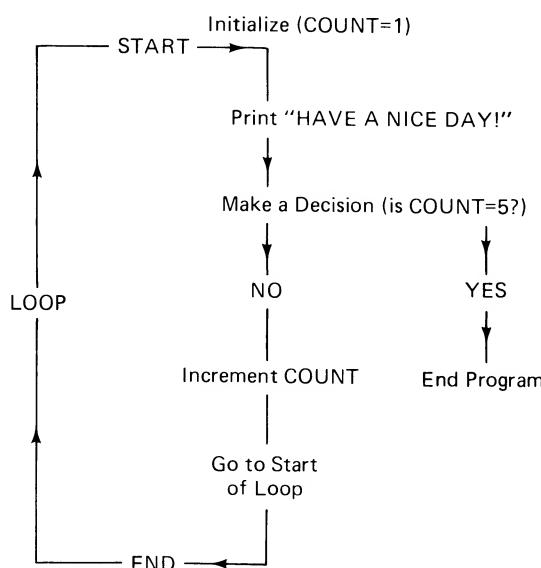
FIGURE 8-2   Structure of a Loop for Program Chapter 8, No. 7

Counters

Notice that in all of the variations of this "NICE DAY" program, the variable COUNT serves only to count how many times we have gone through the loop. We refer to such a variable as a *counter*.

Suppose we would like to find out how many positive integers have squares that are between 300 and 500. Try the following program:

```
100 REM - Chapter 8, No. 8
110 GRAPHICS 0:PRINT
120 REM *** N=NUMBER, S=SQUARE, C=COUNTER ***
130 N=16:REM *** WE REMEMBER THAT 16 SQUARED IS 256 ***
140 S=N*N
150 IF S>=300 AND S<=500 THEN PRINT N,S:C=C+1
160 IF S<=500 THEN N=N+1:GOTO 140
170 PRINT:PRINT
180 PRINT C;" NUMBERS HAVE SQUARES"
190 PRINT "BETWEEN 300 AND 500."
```

As indicated in the REMark statement, the COUNTER in the program is the variable C. Notice that at line 150 if the square of N (variable S) is between 300 and 500, then we do two things:

1. **Print N and S**

2. **Increment the counter**

Because we didn't initialize C, it is automatically zero at the beginning of the program, so that after the first increment (C=C+1), it is one; after the second, two; etc. Thus, it "counts" the number of squares that are between 300 and 500. Note that we still need to test the variable S at line 160 to see if we have gone far enough. Finally, we print the summary at lines 180 and 190. Study the following trace.

TABLE 8-4 Trace of Program Chapter 8, No. 8

Step Number	Statement Number	Values of Variables C	N	S	Decision (Test)	Yes /No	Output	
1	110						(clear screen)	
2	130		16					
3	140			256				
4	150				300<S<500?	NO		
5	160		17		S<=500?	YES		
6	140			289				
7	150				300<S<500?	NO		
8	160		18		S<=500?	YES		
9	140			324				
10	150	1			300<S<500?	YES	18	324
11	160		19		S<=500?	YES		
12	140			381				
13	150	2			300<S<500?	YES	19	381
14	160		20		S<=500?	YES		
15	140			400				
16	150	3			300<S<500?	YES	20	400
17	160		21		S<=500?	YES		
18	140			441				
19	150	4			300<S<500?	YES	21	441
20	160		22		S<=500?	YES		
21	140			484				
22	150	5			300<S<500?	YES	22	484
23	160		23		S<=500?	YES		
24	140			529				
25	150				300<S<500?	NO		
26	160				S<=500?	NO		
27	170						(skip 2 lines)	
28	180						5 NUMBERS HAVE SQUARES.	
29	190						BETWEEN 300 AND 500.	

We hope that you will agree that even though the trace of this program is a bit long, it is an excellent way to see exactly how the counter and the two tests work. Next, we're going to look at a way to eliminate "out of data" error messages.

Flags

Back in Chapter 6, when we discussed the READ and DATA statements, we solved the problem of out of data error messages by adding a FOR and NEXT statement to our programs. We're going to discuss those statements in detail in the next chapter, but now we're going to look at an alternative method.

A *flag* is a code number or string that can be tested in an IF...THEN statement to determine which statement will be executed next. Here is a short program that READs plotting data and uses a flag to see when the data is gone. The plotting technique is the same as in the program Chapter 7, No. 6.

```
100 REM - Chapter 8, No. 9
110 DIM A$(1)
120 GRAPHICS 7:COLOR 1
130 READ A$,X,Y
140 IF A$="S" THEN PRINT "I drew a star!":END
150 IF A$="P" THEN PLOT X,Y:GOTO 130
160 IF A$="D" THEN DRAWTO X,Y:GOTO 130
200 DATA P,80,3
210 DATA D,53,75
220 DATA D,124,31
230 DATA D,36,31
240 DATA D,107,75
250 DATA D,80,3
260 DATA S,0,0
```

Do you see how the flag (string S in line 260) works? As long as a P or a D is read into A$, then the corresponding values for X and Y are used for plotting or drawing. But if S (for stop) is read into A$, then the IF...THEN statement at line 140 ends the program (after printing "I drew a star!"). Notice that P and D in the other DATA statements are also flags. They determine whether to execute the PLOT statement or the DRAWTO statement. Instead of a string, we could use a number for a flag. Make these changes in the program:

```
101 REM - (Addition to Chapter 8, No. 9)
140 IF X=-1 THEN PRINT "I drew a star!":END
260 DATA P,-1,0
```

Note that in each case, we use a value for the flag (S,-1) that is *not a normal data value*. The coordinates for plotting can't be negative, so -1 is a good choice. P and D are the only codes for plotting and drawing, so S will work as a flag. We could, of course, have chosen any other letter of the alphabet, as long as we had used the same letter in the test for the flag at line 140.

While we're at it, let's add a COUNTER to the program to see how many sets (A$,X,Y) of data are processed. You'll need to change lines 100 and 140 and add line 145, as follows:

```
102 REM - (2nd addition to Chapter 8, No. 9)
140 IF X=-1 THEN PRINT "I drew a star
with ";C;" sets of data!":END
145 C=C+1
```

Note that the counter must be incremented *after* we test for the flag, not *before*; otherwise, the flag will be counted as part of the data.

To close our discussion of loops, flags, and counters, here is a program that adds the daily maximum temperatures for a month, finds the average, and prints the results. Study the program to see if you can find the initialization, incrementation, decision, flag, and counter. (Obviously, the program can be modified for another month.)

```
100 REM - Chapter 8, No. 10
110 DIM MONTH$(9)
120 GRAPHICS 0:PRINT
130 TOTAL=0:DAYS=0
140 READ MONTH$
150 READ TEMP
160 IF TEMP=999 THEN 200
170 TOTAL=TOTAL+TEMP
180 DAYS=DAYS+1
190 GOTO 150
200 AVETEMP=TOTAL/DAYS
210 PRINT "The average temperature of the"
220 PRINT DAYS;" days in ";MONTH$;" was ";AVETEMP;" C."
230 DATA FEBRUARY
240 DATA 10,8,4,-1,3,9,15,9,8,10,16,4,-3,-1
250 DATA -3,0,6,15,10,11,15,20,10,8,0,-2,5,-4
260 DATA 999
```

Line 130 is not really necessary because variables are automatically initialized to zero. We have to READ the month (line 140) and the daily temperature (line 150) on separate lines because the month is read only once, but the temperature is read 28 times. Line 170 is a *summing* statement; it adds the daily temperature onto the running total. (This is a technique you will use often.) Line 200 computes the average daily temperature, and then the results are printed. Add these lines to print out the highest and lowest daily temperatures. (Do you see how it works?)

```
101 REM - (Addition to Chapter 8, No. 10
135 HIGH=-100:LOW=100
162 IF TEMP>HIGH THEN HIGH=TEMP
164 IF TEMP<LOW THEN LOW=TEMP
222 PRINT:PRINT "The highest temperature was ";HIGH;"."
224 PRINT "The lowest temperature was ";LOW;"."
```

EXERCISES

1. The statements `130 AGE=AGE+1` and `190 X=X+1` are examples of

    _____ .

2. [( ) True   ( ) False] In ATARI BASIC, numeric variables are
    automatically initialized to 1.

3. In BASIC, the keywords for tests, or decisions, are _____ .

4. If the variable KOUNT is to be used as a counter for the number of
    items that are READ by the BASIC statement `130 READ X` then
    an increment statement that we could use is `170 KOUNT = KOUNT`

    `+` _____ .

5. [( ) True   ( ) False] "Flag" refers to a graphics display in red,
    white, and blue.

6. Write the printout of the following program *without* using the
    computer.

    ```
    100 COUNT=1
    110 READ J,K
    120 PRINT (J+K)/2
    130 IF COUNT=3 THEN STOP
    140 COUNT=COUNT+1
    150 GOTO 110
    160 DATA 2,3,4,5,6,8
    ```

7. Write as a BASIC statement: "If X1 isn't less than Y3, then
    end the 'RUN'." _____ .

8. Correct the following statements as necessary.

    ```
    100 IF A><5 THEN 170

    200 IF A$="JONES THEN 180
    ```

```
300 IF XY<=M+N THEN 190

400 IF P/4 NOT= 7*A THEN 210

500 IF Q/8 = "NO" THEN 200

600 IF P=-5 THEN 220
```

9. Correct the following program so that the titles NAME and AVERAGE will be printed only once. Run the program as it appears below before you correct it.

```
100 DIM NAME$(15)
110 READ NAME$,GR1,GR2,GR3
120 PRINT "NAME","AVERAGE"
130 AVE=(GR1+GR2+GR3)/3
140 PRINT NAME$,AVE
150 GOTO 110
200 DATA SUSIE,53,42,98
210 DATA JOE,77,84,51
220 DATA MARY,98,87,92
230 DATA PETE,78,82,89
```

10. Modify the program in the preceding problem by adding a flag and a test for the flag so that we do not get the "out of data" error message. Let the flag be "ZZZ." Show the lines that must be added below:

_____

_____

11. Consider this problem: The withholding tax for a worker's weekly earnings is computed as follows. For each dependent claimed, subtract $25 from the gross earnings, then take 17% of the result. Given the DATA statements (name, gross earnings, number of dependents) below, write an original program that will compute the taxes and produce the printout as shown. Use a test to see if you need a space after the dollar sign in the tax column. Use a flag to indicate when all of the data is read.

```
200 DATA BROWN,620,3
210 DATA JONES,590,1
220 DATA SMITH,692,4
230 DATA LEE,575,2
240 DATA MILLER,710,0
```

12. Write a trace of the following program without running the program on the computer. The first few lines are done for you on the form on the next page. (Algebra students will recognize that the program tests for the type of roots of a quadratic equation, but you don't need to know that to write the trace.)

```
100 READ A,B,C
110 IF A=0 THEN PRINT:PRINT:END
120 D=B*B-4*A*C
130 IF D<0 THEN PRINT "No real roots":GOTO 100
140 IF D=0 THEN PRINT "Double root":GOTO 100
150 IF D>0 THEN PRINT "Two real roots":GOTO 100
200 DATA 1,3,-4
210 DATA 5,1,3
220 DATA 1,-6,9
230 DATA 0,0,0
```

Step No.	State-ment No.	Values of Variables				Decision (Test)	Yes /No	Output
		A	B	C	D			
1	100	1	3	-4				
2	110					A=0?	NO	
3	120				25			
4	130					D<0?	NO	
5	140					D=0?	NO	
6	150					D>0?	YES	Two real roots
7	100	5	1	3				
8								
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								

13. Suppose that in a program we want the value of the variable ALT to alternate between 0 and 1; that is, at a certain point in the program, we want to change ALT to 0 if it is currently 1, and change it to 1 if it is currently 0. Write a single BASIC statement that will do this.

_____

FOR...NEXT Loops

We have already used FOR and NEXT statements in several programs without discussion or explanation. In this chapter we will explain how they work and give several examples of their use.

Introduction

Let's start by writing the same program in two different ways: (1) with a "normal" loop and (2) with a FOR...NEXT loop.

Two Ways to Write Loops

```
100 REM - Chapter 9, No. 1
110 REM *** NORMAL LOOP ***
120 GRAPHICS 0:PRINT
130 COUNT=1
140 PRINT "PASS THROUGH THE LOOP NUMBER ";COUNT
150 IF COUNT=5 THEN 180
160 COUNT=COUNT+1
170 GOTO 140
180 PRINT:PRINT "END OF PROGRAM"

100 REM - Chapter 9, No. 2
110 REM *** FOR...NEXT LOOP ***
120 GRAPHICS 0:PRINT
130 FOR COUNT=1 TO 5
140 PRINT "PASS THROUGH THE LOOP NUMBER ";COUNT
170 NEXT COUNT
180 PRINT:PRINT "END OF PROGRAM"
```

Immediately, you can see that the FOR...NEXT loop was two statements shorter. Somehow, the functions of initialization, incrementation, decision, and branching were taken care of by the FOR and NEXT statements. Let's consider the statements separately to see how this is done.

The FOR Statement

Here is the general form of the FOR statement with line 130 from the preceding program written underneath it as an example.

```
(line no.) FOR (variable) = (exp 1) TO (exp 2)

    130    FOR    COUNT    =    1    TO    5
```

(Variable) must be a *numeric variable*; and (exp 1) and (exp 2) (expression 1 and expression 2) may be *numerical constants* (such as 50 or 3.14), *numerical variables* (such as Y or TOTAL), or *algebraic expressions* (such as 5*X+2 or TOTAL/3). Note that the variables and expressions in a FOR statement *cannot* be strings. Here are more examples of legitimate FOR statements:

```
110 FOR NUM=1 TO 10
130 FOR COUNT=16 TO 31
170 FOR COUNT=5*X TO 10*X
210 FOR ROW=COL+1 TO 19
300 FOR J=K+1 TO LAST
```

Look again at the general form of the FOR statement and our example from the program, Chapter 9, No. 2:

```
(line no.) FOR (variable) = (exp 1) TO (exp 2)

    130    FOR    COUNT    =    1    TO    5
```

The variable COUNT has an initial value equal to 1. This value is incremented by *one* each time the loop is executed until it is greater than 5, the value of (exp 2). Thus, the FOR statement does the job of initializing, incrementing, and deciding. Think of our example as asking the question: Is the value of COUNT greater than 5? If the answer is *yes*, then the statement immediately following the NEXT statement will be executed next.

The NEXT Statement

The general form of the NEXT statement is

```
(line no.) NEXT (variable)

   130    NEXT    COUNT
```

The variable must be the same one as in the corresponding FOR statement. FOR and NEXT statements *always* go together in pairs; you can't have one without the other. The NEXT statement does the job of branching; it takes the place of the GOTO statement in a normal loop. After the NEXT statement is executed, the FOR statement will be executed again.

The Body of a Loop

The *body* of a loop is the set of statements between the FOR and NEXT statements. In the program, Chapter 9, No. 2, the body is exactly one line: line number 140.

When you write FOR...NEXT loops, you must be sure about what you want in the body of the loop. Type in and run the following program as an example:

```
100 REM - Chapter 9, No. 3
110 GRAPHICS 0:PRINT
120 PRINT "TABLE OF SQUARES":PRINT
130 FOR NUM=1 TO 10
140 PRINT "NUMBER","SQUARE"
150 PRINT NUM,NUM*NUM
160 NEXT NUM
170 PRINT:PRINT "END OF TABLE"
```

Does the display seem satisfactory to you? No? We don't really want the words "NUMBER" and "SQUARE" printed every time we print a number and its square—just at the beginning, before the loop. Can you see how to fix the program? Change line 140 to line 125, then delete line 140, and change the REM statement to Chapter 9, No. 4. The program should look like this:

```
100 REM - Chapter 9, No. 4
110 GRAPHICS 0:PRINT
120 PRINT "TABLE OF SQUARES":PRINT
125 PRINT "NUMBER","SQUARE"
130 FOR NUM=1 TO 10
150 PRINT NUM,NUM*NUM
160 NEXT NUM
170 PRINT:PRINT "END OF TABLE"
```

STEP

We have pointed out that the FOR statement automatically increments the variable *by one* until it is greater than the expression following TO. We can, however, increment by any value by adding the STEP statement to the FOR statement. Change line 130 of your preceding program to read:

```
130 FOR NUM=1 TO 10 STEP 2
```

What happens? You should get only the numbers 1, 3, 5, 7, and 9 and their squares. The next value, 11, is greater than 10, so line 170 is executed next.

We can use the STEP statement to have a loop go backwards, as well as frontwards. Change line 130 as follows:

```
130 FOR NUM=10 TO 1 STEP-1
```

Be careful to make (exp 1) larger than (exp 2) if you use a negative step. This would be a good time to practice changing the expressions in line 130 to see the various possibilities for the FOR...NEXT loop. Here are some suggested changes:

```
130 FOR NUM=1 TO 1
130 FOR NUM=1 TO 10 STEP 5
130 FOR NUM=1 TO 10 STEP-1
130 FOR NUM=10 TO 1
130 FOR NUM=-10 TO -5
130 FOR NUM=1 TO 10 STEP 20
130 FOR NUM=X TO Y
130 FOR NUM=1 TO 10 STEP 1.7
```

Were any of the results surprising to you? You can see that you must be careful to make the expressions consistent with one another.

Timing With FOR...NEXT Loops

We first used FOR...NEXT loops back in Chapter 3 to hold a note in a tune for a particular interval. It takes the computer a certain length of time to execute the statements in the body of a loop; the time depends on the number and complexity of the statements. Even if there is *no* body to the loop (only a FOR statement and a NEXT statement), it still takes time to do the incrementing and testing in the FOR statement. The following program is one that you can use with your watch to determine how long it takes for a "bodiless" FOR...NEXT loop to be executed. Then you will know what constants to put in the FOR statement to hold text on the screen for five seconds, hold a note for three seconds, etc.

First, some suggestions for using the program. Type your response to the prompt "Length of loop", but don't press RETURN until you are ready with the second hand of your watch. (A stop watch would be really helpful here.) When you're ready, press RETURN and start timing. Stop timing when you hear the tone. Our tests showed that a loop of length 4000 takes about 10 seconds to execute; your results may be slightly different. Here's the program:

```
100 REM - Chapter 9, No. 5
110 GRAPHICS 0:PRINT
120 PRINT "Length of loop ";:INPUT L
130 FOR K=1 TO L
140 NEXT K
150 SOUND 0,60,10,10
160 FOR WAIT=1 TO 200:NEXT WAIT
170 END
```

Did you notice that our "wait loop" in line 160 should hold the sound on for about one-half second? The END statement turns the sound off. Can you think of how to make a clock program with a short tone every ten seconds?

Try the following two short programs to see how FOR...NEXT loops can be used with graphics displays. Many variations are possible!

Using FOR...NEXT Loops

```
100 REM - Chapter 9, No. 6
110 GRAPHICS 3:COLOR 1
120 FOR ROW=0 TO 19 STEP 2
130 PLOT 0,ROW:DRAWTO 39,ROW
140 NEXT ROW
```

```
100 REM - Chapter 9, No. 7
110 GRAPHICS 3:COLOR 1
120 FOR COL=0 TO 39 STEP 2
130 PLOT COL,0:DRAWTO COL,23
140 NEXT COL
```

For a little fun, add the following lines to the preceding program:

```
101 REM - (Adition to Chapter 9, No. 7)
115 COLOR 1
145 FOR WAIT=1 TO 100:NEXT WAIT
150 COLOR 2
```

```
160 FOR COL=38 TO 0 STEP -2
170 PLOT COL,0:DRAWTO COL,23
180 NEXT COL
185 FOR WAIT=1 TO 100:NEXT WAIT
190 GOTO 115
```

Now, try this new program, which plays all the possible pure tones:

```
100 REM - Chapter 9, No. 8
110 FOR TONES=0 TO 255
120 SOUND 0,TONES,10,10
130 NEXT TONES
```

If the tones "play" too fast, then insert this line:

```
125 FOR WAIT=1 TO 25:NEXT WAIT
```

Do you remember that we can print part of a string? For example, A$(2,5) refers to the second through fifth columns of the string A$. The following program uses this idea to print an increasing number of blank spaces at the beginning of a line.

```
100 REM - Chapter 9, No. 9
110 DIM BLANK$(10),X$(10)
120 BLANK$="          ":X$="XXXXXXXXXX"
130 GRAPHICS 0:PRINT
140 FOR LINE=1 TO 10
150 PRINT BLANK$(1,LINE);X$
160 NEXT LINE
```

If you have a slightly "weird" sense of humor, add the following line to the program:

```
170 PRINT:PRINT BLANK$;"   PISA"
```

Now, see if you can make the tower lean in the other direction. Hint: Do you remember how to make a loop go backwards?

You might want to see some of the different colors that the ATARI can produce. Here is a program that changes the color of the background.

```
100 REM - Chapter 9, No. 10
110 GRAPHICS 1
120 FOR HUE=0 TO 15
130 SETCOLOR 4,HUE,4
```

```
140 PRINT "This is hue number ";HUE;"."
150 FOR WAIT=1 TO 100:NEXT WAIT
160 NEXT HUE
```

If you would like the colors brighter or darker, change the last 4 in line 130. (Remember, this is the luminance value—it can be an *even* integer between 0 and 14.) Can you think of how you might modify this program to look at all of the luminances of a single color—say, red?

Getting Out of FOR...NEXT Loops

A FOR...NEXT loop need not be completed. We may want to have a decision in the body of the loop that might cause us to leave the loop before the maximum value of the loop variable has been reached. One common example is to "search" a string for a certain character, as in the following program. Please note that lines 160 and 180 each occupy two screen lines. Don't press the RETURN key until you have finished the second screen line.

```
100 REM - Chapter 9, No. 11
110 DIM WORD$(10)
120 GRAPHICS 0:PRINT
130 PRINT "Type a 'word' of exactly"
140 PRINT "10 letters ":INPUT WORD$
150 FOR K=1 TO 10
160 IF WORD$(K,K)="A" THEN PRINT "Lett
er number ";K;" is an A.":END
170 NEXT K
180 PRINT:PRINT WORD$;" does not have
an A in it."
```

Do you see that WORD$(K,K) refers to the Kth letter of WORD$? For example, WORD$(5,5) refers to the fifth letter, WORD$(7,7) refers to the 7th letter, etc. When an "A" is found in the word, then the results are printed and the program ends. If the word does not contain an "A," then the loop is executed 10 times, and then the message at line 180 is printed.

Nested FOR...NEXT Loops

If we place one loop inside another loop, we call the loops *nested loops*. We have already done this when we used "wait" loops inside other loops. Try this example of nested FOR...NEXT loops that fills the screen with squares:

```
100 REM - Chapter 9, No. 12
110 GRAPHICS 3:COLOR 1
```

```
120 FOR COL=1 TO 39 STEP 2
130 FOR ROW=1 TO 19 STEP 2
140 PLOT COL,ROW
150 NEXT ROW
160 NEXT COL
```

Let's consider just what happens in the loops. We start with the value of COL equal to 1. Then the entire inner loop with the variable ROW is completed. That is, we plot the points **(1,1)**, **(1,3)**, **(1,5)**, ..., **(1,17)**, **(1,19)**. We have finished the inner loop, so we go back to the outer loop; this time with the value of COL equal to 3 (remember the increment is 2). Then the inner loop is completed again, COL is incremented, then the inner loop is completed again, etc. Run the program again and see if you can see this process. When you feel comfortable with it, try the next program (a modification of the program Chapter 9, No. 10), which shows all colors and all luminances.

```
100 REM - Chapter 9, No. 13
110 GRAPHICS 1
120 FOR HUE=0 TO 15
130 FOR LUM=0 TO 14 STEP 2
140 SETCOLOR 4,HUE,LUM
150 PRINT "HUE = ";HUE,"LUMINANCE = ";LUM
160 FOR WAIT=1 TO 200:NEXT WAIT
170 NEXT LUM
180 NEXT HUE
```

This is an example of *triple* nested FOR...NEXT loops. First a hue is picked, then a luminance, then we wait for a while, then a new luminance, then wait, etc. Notice that the NEXT statements *always* occur in the *reverse order* of the FOR statements, so that each loop is completely contained in the one outside it. The next program prints part of the multiplication table (the part we have trouble with!).

```
100 REM - Chapter 9, No. 14
110 GRAPHICS 0:POKE 752,1
120 PRINT "MULTIPLICATION TABLE FOR 6-7-8-9"
130 FOR K=6 TO 8 STEP 2
140 PRINT
150 FOR N=1 TO 10
160 PRINT K;" x ";N;" = ";K*N,
170 PRINT K+1;" x ";N;" = ";(K+1)*N
180 NEXT N
190 NEXT K
200 GOTO 200
```

Note that when the value of K is 6 and the value of N is 1, then lines 160 and 170 print the following:

```
6 x 1 = 6              7 x 1 = 7
```

Do you see the comma at the end of line 160? It keeps the cursor on the same line so that we can have the table for 7 opposite the table for 6, and the table for 9 opposite the table for 8.

Here is an example of nested loops with READ and DATA statements. We have tried to show the beginning and end of both loops: the "outer" (PLAYER) loop, and the "inner" (GAME) loop. This is a good technique for your own programs.

```
        100 REM - Chapter 9, No. 15
        110 DIM PLAYER$(20)
        120 GRAPHICS 0:PRINT
        130 PRINT "BASKETBALL STATISTICS FOR 8 GAMES"
        140 PRINT "=================================="
        150 PRINT
        160 PRINT "PLAYER'S NAME","TOTAL ","AVERAGE"
        170 PRINT ,,"POINTS"
        180 PRINT "--------------","-------","-------"
        185 REM *** PROCESS-5-PLAYERS LOOP ***
----------------------------
¦       190 FOR PLAYER=1 TO 5
¦       200 READ PLAYER$
¦       210 TOTAL=0
¦       215 REM *** GAMES-FOR-EACH-PLAYER LOOP ***
¦       --------------------
¦   ¦   220 FOR GAME=1 TO 8
¦   ¦   230 READ POINTS
¦   ¦   240 TOTAL=TOTAL+POINTS
¦   ¦   250 NEXT GAME
¦       --------------------
¦       255 REM *** COMPUTE AVERAGE, PRINT RESULTS ***
¦       260 AVERAGE=TOTAL/8
¦       270 PRINT PLAYER$,TOTAL,AVERAGE
¦       280 NEXT PLAYER
----------------------------
        290 DATA GARY GUARD,5,8,3,15,4,12,7,4
        300 DATA HENRY HOTSHOT,12,10,8,6,14,12,11,7
        310 DATA FRED FORWARD,4,0,8,2,0,4,5,3
        320 DATA RICKY REBOUND,7,8,0,0,5,1,2,6
        330 DATA TOM TIPSHOT,10,12,8,20,14,19,13,9
```

Note that at line 210, TOTAL must be *reset* to zero for each player; otherwise, Tom Tipshot gets all the points!

Here's another example of a multiplication table with a more "normal" format:

```
        100 REM - Chapter 9, No. 16
        110 GRAPHICS 0:PRINT
```

**Printing Techniques
With FOR...NEXT
Loops**

```
120 PRINT "MULTIPLICATION TABLE"
130 PRINT "===================="
140 PRINT:PRINT
150 PRINT " x ¦ 1    2    3    4    5"
160 PRINT "---¦----------------------"
```
```
170 FOR ROW=10 TO 20
180 PRINT ROW;" ¦ ";
```
```
190 FOR COL=1 TO 5
200 PRINT ROW*COL;"   ";
210 NEXT COL
```
```
220 PRINT
230 NEXT ROW
```

If you were careful with the spacing, the rows and columns all lined up. (If they didn't, check the list of your program against the list above and make corrections.) Note the following in this program: First, at line 180, the number of the row and a vertical dividing line are printed. The semicolon at the end of the line keeps the cursor on the same line. Second, when the inner (COL) loop is finished, the cursor is still on the same line because of line 200. To move the cursor to the next line, we need the PRINT statement at line 220. See what happens if you delete this line.

Now try the following two programs to see what happens if we cross loops.

```
100 REM - Chapter 9, No. 17
110 GRAPHICS 0:PRINT
112 PRINT "OUTER","INNER"
114 PRINT "-----","-----"
```
```
120 FOR OUTER=1 TO 3
```
```
130 FOR INNER=1 TO 4
140 PRINT OUTER,INNER
150 NEXT INNER
```
```
160 PRINT
170 NEXT OUTER
```

```
100 REM - Chapter 9, No. 18
110 GRAPHICS 0:PRINT
112 PRINT "OUTER","INNER"
114 PRINT "-----","-----"
```

```
!      120 FOR OUTER=1 TO 3
!    ---------------------
!  !   130 FOR INNER=1 TO 4
!  !   140 PRINT OUTER,INNER
!  !   150 NEXT OUTER
---!---------------------
   !   160 PRINT
   !   170 NEXT INNER
   ---------------------
```

Did you get an ERROR- 13 AT LINE 170 message when you ran the second version? That means "no matching FOR statement." In this case, it is the result of crossing the loops.

RULE: Never cross loops!

This would be a good place to mention a second rule.

RULE: Never enter the body of a loop with a GOTO statement from outside the loop— error 13 will be the result.

We close this chapter with a program that will print out all of the different permutations of the letters in the word "LOVE". Notice the quadruple nested loops!

```
100 REM - Chapter 9, No. 19
110 DIM A$(4)
120 A$="LOVE"
130 GRAPHICS 0:PRINT
140 FOR A=1 TO 4
150 FOR B=1 TO 4
160 IF B=A THEN 270
170 FOR C=1 TO 4
180 IF C=A THEN 260
190 IF C=B THEN 260
200 FOR D=1 TO 4
210 IF D=A THEN 250
220 IF D=B THEN 250
230 IF D=C THEN 250
240 PRINT A$(A,A);A$(B,B);A$(C,C);A$(D,D)
250 NEXT D
260 NEXT C
270 NEXT B
280 NEXT A
```

The IF statements are necessary to be sure we don't pick a letter that has already been picked. You can try another word with 4 letters in line 120.

EXERCISES

1.   What error message will the following one-liner produce?

```
150 FOR K=1 TO 5:PRINT K:NEXT M
```

2.   [( )True   ( )False]   In ATARI BASIC, STEP may be a variable.

3.   If the STEP option is left out of a FOR statement, the step is auto-matically equal to _____ .

4.   Describe the printout of the following program without using the computer.

```
100 FOR S=1 TO 10
110 S=S*S
120 PRINT S
130 NEXT S
```

5.   [( )True   ( )False]   The FOR statement in a FOR...NEXT loop takes the place of the GOTO statement in an IF...THEN loop.

6.   Write the printout of the following program without using the computer.

```
100 FOR NUM=11 TO 15
110 READ J,K
120 PRINT NUM,J,K,(J+K)/2
130 IF J*J-K*K>50 THEN END
140 NEXT NUM
150 DATA 4,3,11,9,-12,5
160 DATA 7,3,15,20,-5,0
```

7.   The statements between the FOR and NEXT statements of a FOR...NEXT loop are known as the _____ .

8.   [( ) True   ( ) False]   In ATARI BASIC, STEP may *not* be a negative decimal.

9. In the following exercises, write the set of values for the variable and an appropriate NEXT statement.

| Basic Statement	Set of Values for the Variable	Next Statement
100 FOR K=4 TO 8	4, 5, 6, 7, 8	140 NEXT K
110 FOR Y=-2 TO 1	_____	_____
120 FOR M=4 TO 10 STEP 2	_____	_____
130 FOR P2=0 TO 10 STEP 3	_____	_____
140 FOR L3=5 TO 0 STEP -1	_____	_____
150 FOR K=10 TO 1 STEP -1	_____	_____
160 FOR K=-2 TO 2 STEP 0.5	_____	_____
170 FOR A=1 TO 0 STEP -0.2	_____	_____
180 FOR P=4 TO 6.5 STEP 0.6	_____	_____
190 FOR Z3=8 TO 4 STEP -2.5	_____	_____
200 FOR N=P TO P*2 (LET P=5)	_____	_____

10. If one FOR...NEXT loop is placed inside another FOR...NEXT loop, then we refer to the loops as _____ loops.

11. Use a digital watch to time each of the following programs. Time from the instant you press the RETURN key after typing "RUN" until the word "READY" appears on the screen.

```
100 L=1000        ¦   100 L=1000          ¦   100 L=1000
110 FOR N=1 TO L  ¦   110 FOR N=1 TO L    ¦   110 FOR N=1 TO L
120 NEXT N        ¦   120 PRINT N         ¦   120 PRINT N,N*N,N^3
                  ¦   130 NEXT N          ¦   130 NEXT N
```

_____ seconds   _____ seconds   _____ seconds

12. Write an original program using FOR...NEXT loops (and/or nested FOR...NEXT loops) that will fill the GRAPHICS 3 screen with alternating orange and green horizontal stripes, then after a short pause, fill the screen with alternating blue and white vertical stripes. Be sure the stripes are drawn in alternating order: orange, green, orange, green, etc.

13. Write an original program that will play three octaves of a musical scale. Turn the sound off for a short interval between the notes. READ the pitch values (see Appendix 2) from DATA statements.

14. Write an original program using FOR...NEXT loops (and/or nested FOR...NEXT loops) that will fill the GRAPHICS 0 screen with the pattern below. Use some of the printing techniques described in the section, Using FOR...NEXT Loops. Be sure to fill all 24 lines of the screen, and suppress the cursor with a POKE 752,1 statement. You will need to find a way to keep the cursor at position **38,23** at the end of the program. Note that the normal margin of 2 characters on the left side is repeated on the right side.
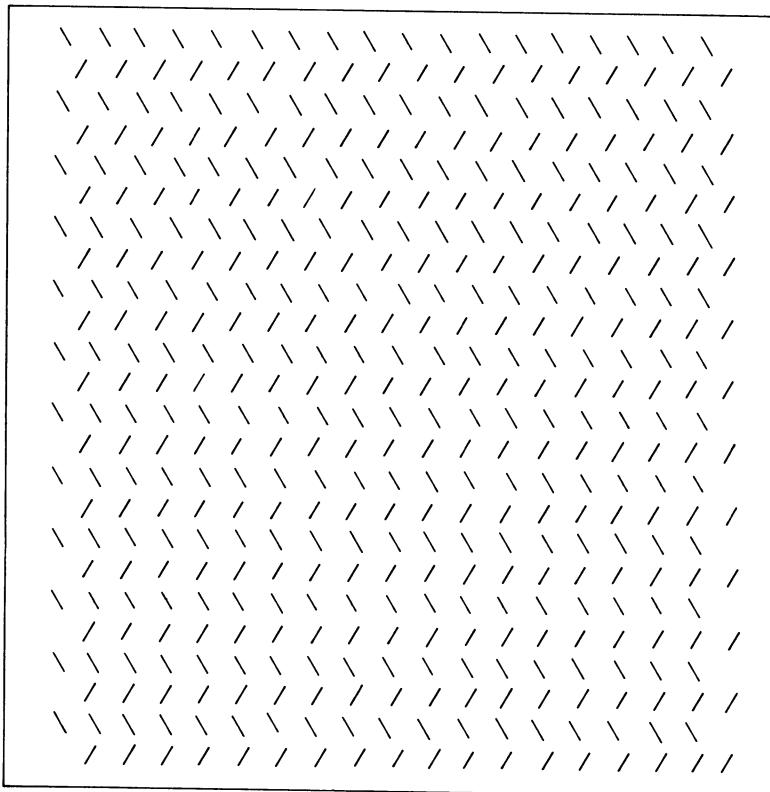
Flowcharts

We have reached a point in our command of BASIC where we are able to put together rather complex programs. In order to avoid mistakes in logic in writing programs, it is often helpful to make a diagram that shows how the various parts of a program are related. Such a diagram is known as a *flowchart*. It is part of the planning for a computer solution to a problem and is usually drawn before the actual program is written. It may also serve as part of the *documentation* of the program.

A Sample Flowchart

Here is a simple program (Chapter 8, No. 4). On the following page we have shown the flowchart.

```
100 REM - Chapter 10, No. 1 (Chapter 8, No. 4)
110 GRAPHICS 0:PRINT
120 COUNT=10
130 PRINT "HAVE A NICE DAY!"
140 IF COUNT>11 THEN END
150 COUNT=COUNT+1
160 GOTO 130
```

Note that the statements in the flowchart boxes are not BASIC statements but simple English statements of what we want the computer to do at a particular point in the program. Now let's consider the shapes of the boxes in a flowchart.
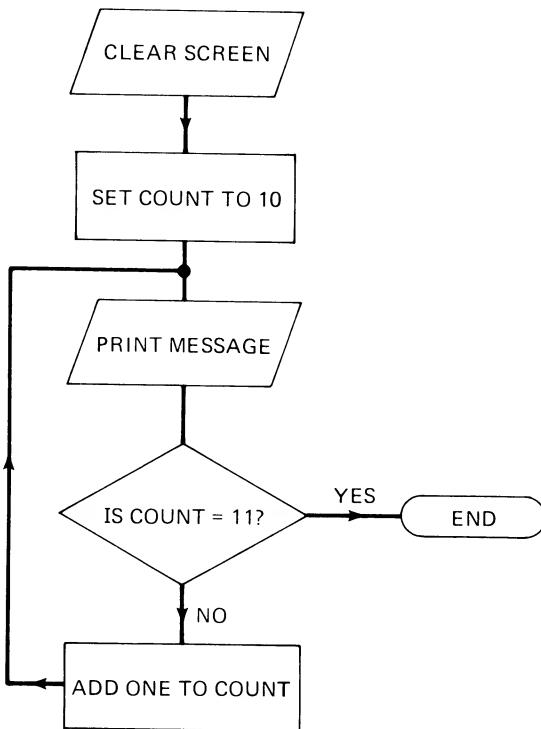
FIGURE 10-1   Flowchart for Program Chapter 10, No. 4

| Flowcharting Symbols | The symbols shown on the next page are in general use for flow-charting for all types of computers and for most programming languages. Inexpensive flowcharting templates, which make it easy to draw the symbols, are readily available. |

Reading Flowcharts

Here are two runs of a program. The corresponding flowchart (Figure 10-3) appears on page 130. See if you can make the connection between the two.

```
RUN

How old are you
?16
You can't vote yet.

READY
■
```

```
RUN

How old are you
?21
You can vote.

READY
■
```

INPUT/OUTPUT

REPRESENTS:
Data or information going in or out
of the computer.
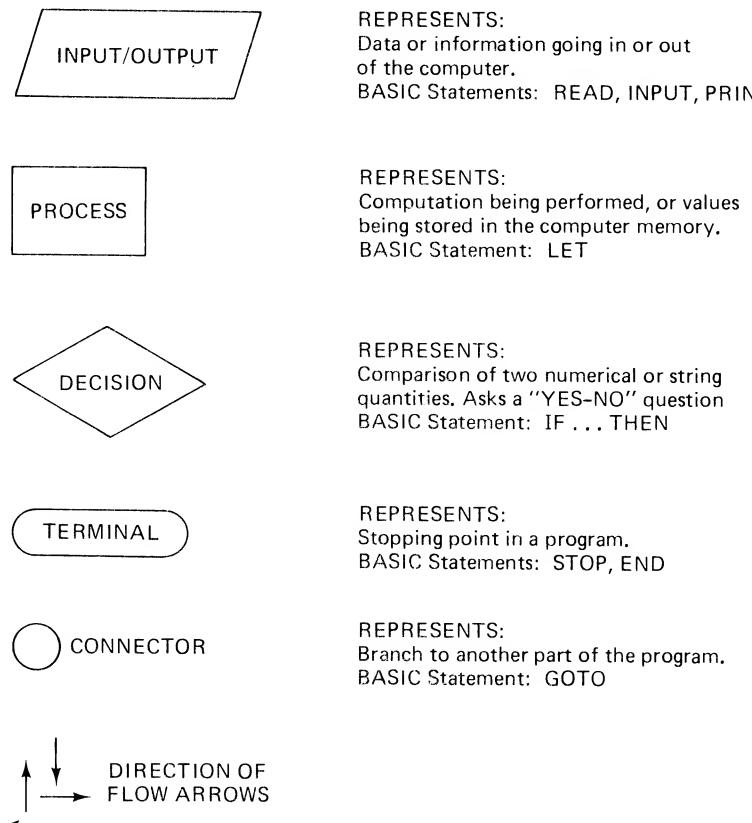BASIC Statements: READ, INPUT, PRINT

PROCESS

REPRESENTS:
Computation being performed, or values
being stored in the computer memory.
BASIC Statement: LET

DECISION

REPRESENTS:
Comparison of two numerical or string
quantities. Asks a "YES–NO" question
BASIC Statement: IF ... THEN

TERMINAL

REPRESENTS:
Stopping point in a program.
BASIC Statements: STOP, END

CONNECTOR

REPRESENTS:
Branch to another part of the program.
BASIC Statement: GOTO

DIRECTION OF
FLOW ARROWS

FIGURE 10-2   Flowcharting Symbols

Normally in a flowchart the direction of flow is from top to bottom and from left to right. We use the direction of flow arrows to show a contrary direction, as at the bottom of the flowchart in Figure 10-3. Note that we used the terminal symbol to indicate the beginning of the flowchart. This is optional.

Here are some sample runs of a program that tests a pair of coordinates input by the user to see if they represent a point that can be plotted in GRAPHICS 3. Note the horizontal placement of the two decision boxes in the flowchart (Figure 10-4) on page 131.

```
RUN

Type the coordinates of
a point (EX.: 5,10).
?25,5

25,5 can be plotted
in GRAPHICS 3.

READY
```

```
RUN

Type the coordinates of
a point (EX.:  5,10).
?5,25

5,25 can't be plotted
in GRAPHICS 3.

READY
```
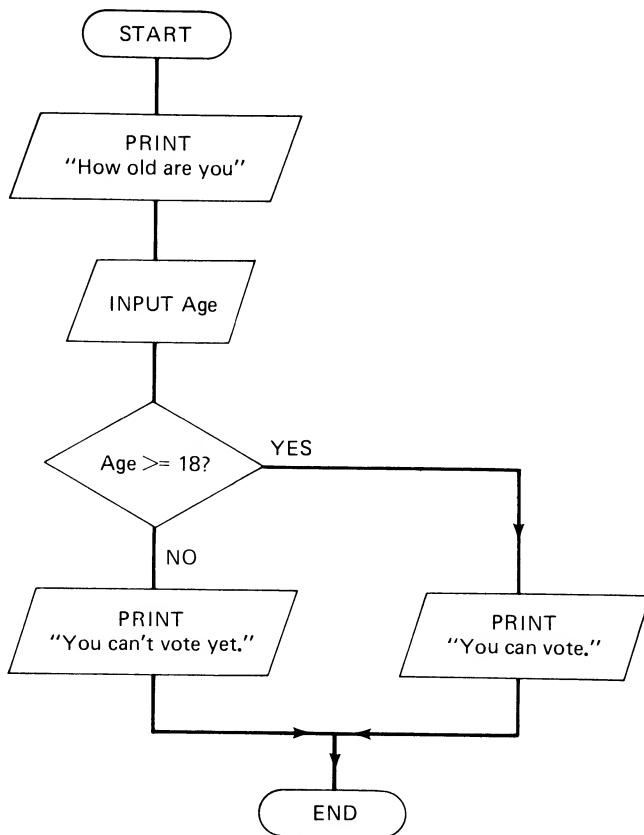
FIGURE 10-3   Flowchart for Voting Program

The following program reads four names and scores and then prints them out with the average of the scores. A flag is used to indicate the end of the data. Note how the connector (A) is used in the flowchart (Figure 10-5) on page 132.

```
RUN

NAME          SCORE
----          -----
Joan           80
Fred           72
Bill           94
Lisa           86

AVERAGE:       83

READY
■
```

Here is another example of a flowchart (Figure 10-6 on page 133) using the connector symbol to indicate a loop. Compare it with the flowchart in Figure 10-1, where we simply drew a line to
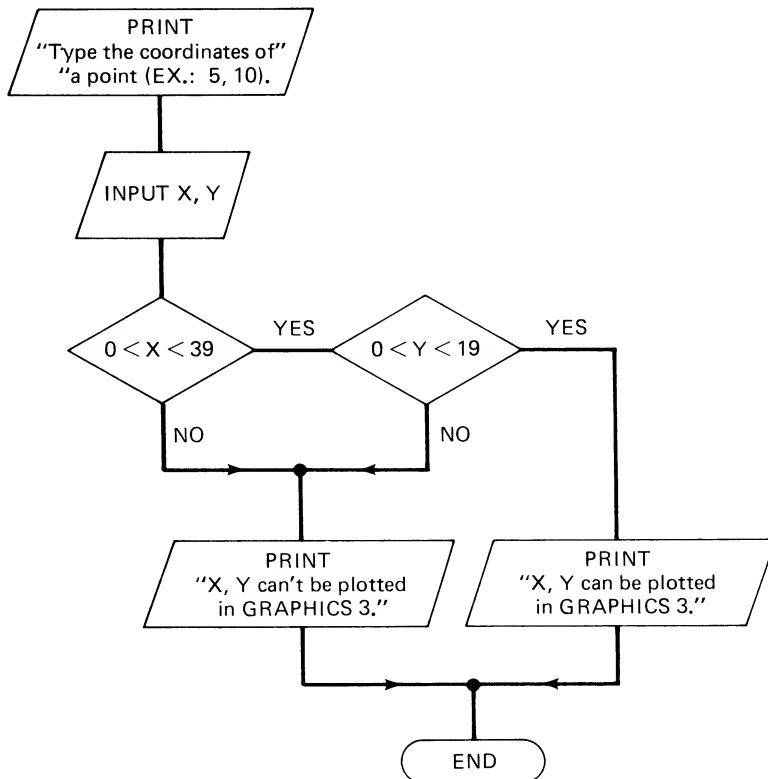
FIGURE 10-4   Flowchart for Coordinates Program

indicate the loop. Using the connector symbol usually makes the flowchart look less cluttered, and therefore, easier to read. Remember that using the connector symbol is an alternate method of showing loops; you may draw lines to show the loop if you prefer.

As you study the flowchart in Figure 10-6, cover the printout given at the right of the chart and see if you can tell what it is going to be.

If we are given an adequate flowchart of a problem to be solved, then it should be a relatively easy matter to write a BASIC program from the flowchart. Following are the programs to accompany the following two flowcharts (Figures 10-5 and 10-6). Type in and run the programs and compare them with the flowcharts. Try to determine how each line of the program shows up in the flowchart in Figure 10-5 and how each symbol in the flowchart is reflected in the program.

Writing Programs From Flowcharts

```
100 REM - Chapter 10, No. 2
110 DIM NAME$(4)
```
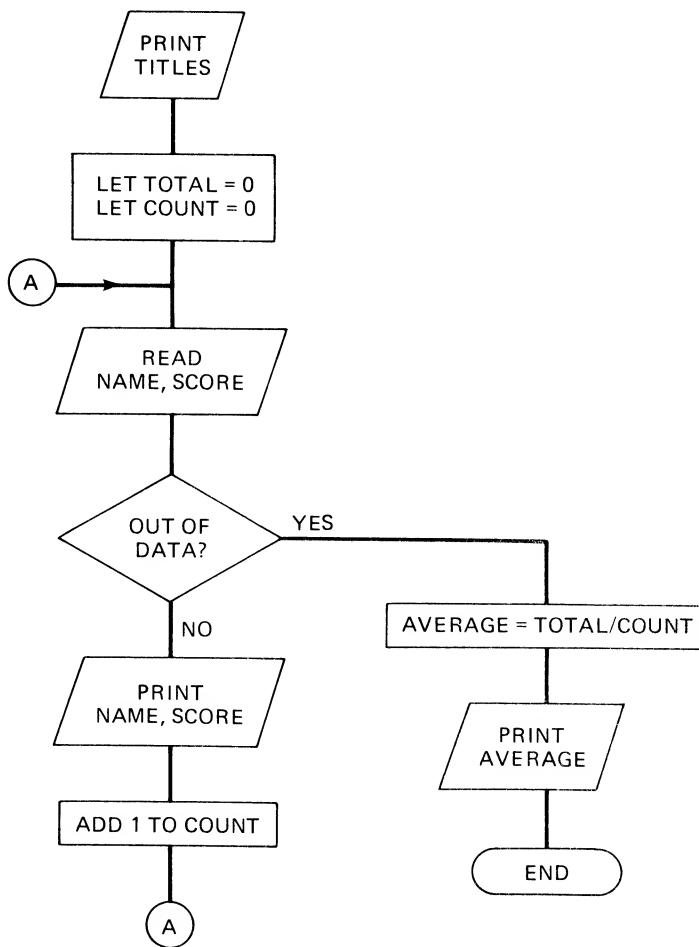
FIGURE 10-5 **Flowchart of Names/Scores Program**

```
120 GRAPHICS 0:PRINT
130 PRINT "NAME","SCORE"
140 PRINT "----","-----"
150 READ NAME$
160 READ SCORE
170 IF NAME$="ZZZ" THEN 220
180 PRINT NAME$,"   ";SCORE
190 TOTAL=TOTAL+SCORE
200 COUNT=COUNT+1
210 GOTO 150
220 AVERAGE=TOTAL/COUNT
230 PRINT: PRINT "AVERAGE:","   ";AVERAGE
240 DATA Joan,80,Fred,72,Bill,94,Lisa,86,ZZZ,0
```

Notice that we don't put REMark statements in the flow-
chart, and conversely, we don't usually need to initialize vari-
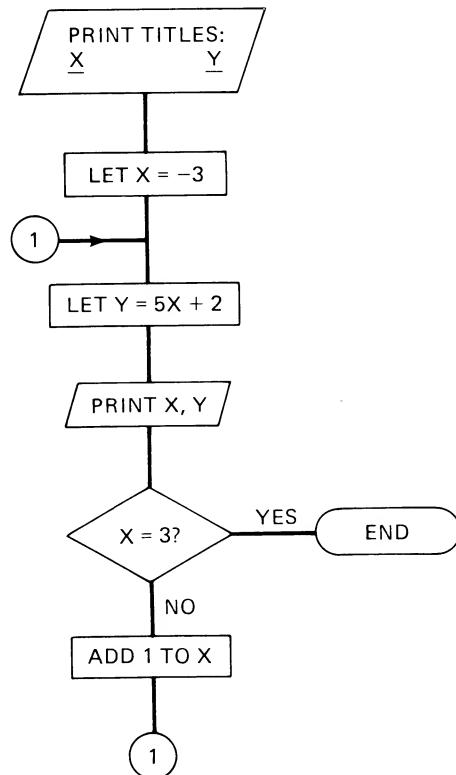
FIGURE 10-6 Mystery Flowchart and Printout

ables to 0 in the program. Sometimes one box in the flowchart takes more than one line of code in the program. For example, the box containing the instruction READ NAME, SCORE corresponds to lines 150 and 160 in the program. We answered the question "Out of data?" by checking a flag at the end of the data list. We always indicate an END in the flowchart, but the program may not actually contain an END statement.

```
100 REM - Chapter 10, No. 3
110 GRAPHICS 0:PRINT
120 PRINT "X","Y"
130 PRINT "==","==="
140 X=-3
150 Y=5*X+2
160 PRINT X,Y
170 IF X=3 THEN END
180 X=X+1
190 GOTO 150
```

You can see that this second program is almost an exact line by line translation from the flowchart in Figure 10-6. A good test

of any one of your flowcharts would be to see if one of your class-mates could write the program from it!

Problem Solving With Flowcharts

Suppose that we would like to design a program to determine whether or not three integers that the user types into the computer could represent the lengths of the sides of a right triangle. For example, 3, 4, and 5, could because 3 squared plus 4 squared equals 5 squared (The Pythagorean Theorem). On the other hand, 5, 7, and 13 wouldn't even represent the lengths of the sides of a triangle because $5 + 7 < 13$. (In a triangle, the sum of the lengths of the two shorter sides is greater than the length of the longest side.)

Before we make a flowchart, we might make a preliminary list of the steps in the program:

1. Clear the screen.
2. Print a prompt to the user.
3. Accept the user's input.
4. Test to see if we have a triangle.
    a. Find the longest side.
    b. Compare with sum of two remaining sides.
5. Test to see if we have a right triangle.
    a. Find sum of squares of shortest sides.
    b. Compare sum with square of longest side.
6. Print results:
    a. No triangle
    b. Right triangle
    c. Triangle.

Step 4 is really the key to the problem; the flowchart for just this step is given below. Suppose that the user inputs the numbers A, B, and C. We need to find the largest of these. Let L represent the largest number, and let S1 and S2 represent the two smaller numbers. We'll start by letting L equal A; that is, we'll assume that A is the largest. Then test to see if B is larger than L. If so, replace L with B. Similarly, test to see if C is larger than L. If so, replace L with C. Then go either to step 5 or to step 6b. In either case, work with the variables L, S1, and S2. The completion of the entire flowchart is given as an exercise.

To draw the complete flowchart, you may draw each part separately and use the connector circles, as we did in Figure 10-7.
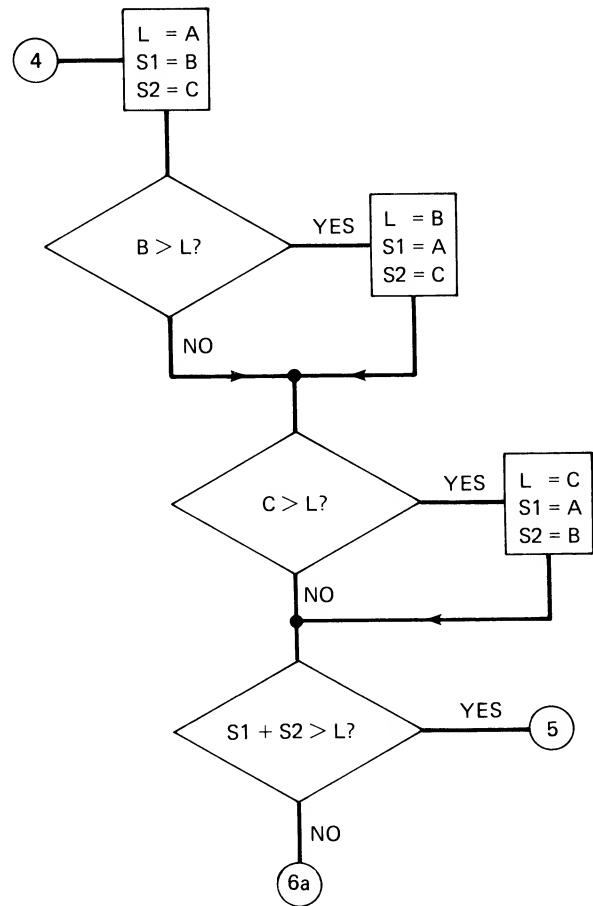
FIGURE 10-7   Flowchart for Right Triangle Problem

EXERCISES

In problems 1. — 10., indicate the letter of the flowcharting symbol that could represent the given basic statement. The symbols are:
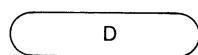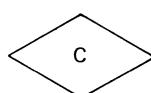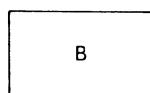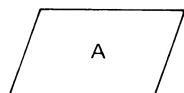
A   B   C

D   E   F — none

1. INPUT A,B,C   1. _____

2. PRINT "Age = ";AGE   2. _____

3. Y=A*X*X+B*X+C   3. _____

4. STOP   4. _____

5. REM - Chapter 7, No. 4   5. _____

6. GOTO 210   6. _____

7. READ NAME$ 7. _____

8. DIM SCHOOL$(30)   8. _____

9. NEXT K   9. _____

10. IF ANSWER$="YES" THEN 320   10. _____

Problems 11. - 15. True or False.

11. Each line in a program corresponds to one   11. _____
    symbol in the flowchart for the program.

12. The flowcharting symbol that corresponds   12. _____
    to the STOP statement is an octagon.

13. The normal order of flow in a flowchart is   13. _____
    from top to bottom and from left to right.

14. The flowcharting symbol that corresponds to   **14.** _____
    the DATA statement is the rectangle.

15. The statement `FOR K=1 TO 10` would be repre-   **15.** _____
    sented by two flowcharting symbols.

16. Write the program for the flowchart given in Figure 10-4. Test the
    program by running it and comparing the results to the sample runs
    on page 129.

17. Complete the flowchart for the right triangle problem on page 134.
    Part 4 is already done for you in Figure 10-7. After you have drawn
    the flowchart, write the program for the solution to the problem.

11

Writing Understandable Programs

Introduction

In the last chapter we looked at methods of diagramming the logical structure of a program. After we have made a flowchart for a program, we have to write the code (BASIC instructions) so that we can understand it ourselves, and so that someone else can understand it. In this chapter we'll discuss some techniques for making program lists easier to understand and some techniques for making the screen output of a program more attractive and readable.

The REMark Statement

We have used the REMark statement since Chapter 3. We can use the REMarks in three ways in a program:

1. **To identify the program by giving the name of the program, the author's name, the date the program was written, and a short description of what the program does.**

2. **To identify the individual sections of a program and to indicate what the next section of the program does.**

3. **To describe the functions of the variables used in the program, and to indicate which actual colors correspond to the SET-COLOR and COLOR numbers used in the program.**

Effective use of REMark statements makes it much easier to modify a program and also decreases the need for extra written documentation of the program. Some programmers write a REMark statement for every line of the program—no further

documentation is necessary! That might make the program excessively long, but it is certainly better to have too many REMarks in a program than to have too few.

Using REMarks for Program Identification

The first lines of a program should always contain some identification. If the program is stored on disk, then the name of the program and the disk number (or name) should be given. Here are some sample identifications for the beginning of programs:

```
100 REM - LOOPING
101 REM - Disk #3
102 REM - James P. Smith
103 REM - Wilson High School
104 REM - 23 MAR 1983
105 REM - Demonstration of nested loops


100 REM ****************************
101 REM **                       **
102 REM **        GRAPHLIN        **
103 REM **                       **
104 REM ** Paul Brown  - 19 JAN 82 **
105 REM **                       **
106 REM ** Graph of linear equation **
107 REM **                       **
108 REM ****************************


100 REM # COLORPIX           #
101 REM # Disk "Graphics #5"  #
102 REM # Randolph Pixel      #
103 REM # Mr. Lee - Period 2  #
104 REM # Graphics 7 Pictures #


100 REM   //      BAKESALE      \\
101 REM   //    Disk "Clubs 3"   \\
102 REM  <<      Susi Rahrah      >>
103 REM   \\ Pd. 4 - Miss Irata //
104 REM   \\     3 APR 83       //
```

Your teacher has probably given you a required form for REMark statements at the beginning of a program. Remember that the function of these statements is to inform subsequent readers (including the original programmer!) about the program. We can guarantee that you will forget many details of a program very quickly after you have written it. A REMark statement is well worth the time it takes to type it into a program.

Using REMarks to Describe Sections of a Program

Here's a program again from Chapter 8. It averages the temperatures for the days of the month of February.

```
100 REM - Chapter 11, No. 1
101 REM - (Chapter 8, No. 10)
110 DIM MONTH$(9)
120 GRAPHICS O:PRINT
130 TOTAL=0:DAYS=0
140 READ MONTH$
150 READ TEMP
160 IF TEMP=999 THEN 200
170 TOTAL=TOTAL+TEMP
180 DAYS=DAYS+1
190 GOTO 150
200 AVETEMP=TOTAL/DAYS
210 PRINT "The average temperature of the"
220 PRINT DAYS;" days in ";MONTH$;" was ";AVETEMP;" C."
230 DATA FEBRUARY
240 DATA 10,8,4,-1,3,9,15,9,8,10,16,4,-3,-1
250 DATA -3,0,6,15,10,11,15,20,10,8,0,-2,5,-4
260 DATA 999
```

Now let's add several REMarks to the program:

```
100 REM - Chapter 11, No. 1
101 REM - Find the average temperature
102 REM - for the month of February
103 REM - J. Reisinger - 1983
104 REM --------------------------
109 REM - Dimension string for month name
110 DIM MONTH$(9)
119 REM - Clear screen and skip a line
120 GRAPHICS O:PRINT
128 REM - Set tally for temperature and
129 REM - counter for days to zero.
130 TOTAL=0:DAYS=0
140 READ MONTH$
150 READ TEMP
159 REM - Watch for flag at end of data
160 IF TEMP=999 THEN 200
169 REM - Add temperature to total
170 TOTAL=TOTAL+TEMP
179 REM - Add 1 to number of days
180 DAYS=DAYS+1
189 REM - Read the next temperature
190 GOTO 150
199 REM - Find the average of the temperatures
200 AVETEMP=TOTAL/DAYS
209 REM - Print the average
210 PRINT "The average temperature of the"
220 PRINT DAYS;" days in ";MONTH$;" was ";AVETEMP;" C."
230 DATA FEBRUARY
239 REM - Data:  Temperatures for month of February
240 DATA 10,8,4,-1,3,9,15,9,8,10,16,4,-3,-1
250 DATA -3,0,6,15,10,11,15,20,10,8,0,-2,5,-4
260 DATA 999
```

An alternative to adding a REMark before each line or group of lines is to place the REMark on the same line, separated by a colon (:). For example, instead of

```
109 REM - Dimension string for month name
110 DIM MONTH$(9)
119 REM - Clear screen and skip a line
120 GRAPHICS 0:PRINT
128 REM - Set tally for temperature and
129 REM - counter for days to zero.
130 TOTAL=0:DAYS=0
```

We could write:

```
110 DIM MONTH$(9):REM - String  for name of month
120 GRAPHICS 0:PRINT:REM - Clear screen, skip line
130 TOTAL=0:DAYS=0:REM - Set counters to zero
```

For another example of using REMark statements to describe the sections of a program, refer to the "Singing Practice" program at the end of Chapter 3. In this program, the REMark statements are particularly helpful because they help to correct typing errors. For example, if you type in the program and find that the eyes cross when you run the program, then the problem must lie in one of the sections labeled "Draw pupils" or "Move pupils." You can imagine how helpful REMark statements could be in a long program.

Describing the Variables in a Program

One extremely useful feature of ATARI BASIC is that variable names may contain as many as 120 characters. The variable name can suggest what is stored in that variable; for example, the variable names AREA, WAGES, AVERAGE, AGE, FEET and POUNDS are self-explanatory. Using such names makes the program list much easier to read. However, it may still be necessary to indicate what is stored in a particular variable name. A useful practice is to list and describe all of the variables used in a program somewhere near the beginning of the program.

The program on the next page, Chapter 11, No. 2, is an example of using REMark statements in this way. The program draws some large letters on the GRAPHICS 7 screen. Each letter is designed from a 7-row by 4-column grid as shown in Figure 11-1. At the right, the letter E is formed from the grid. In the program, each letter is drawn by plotting the upper left-hand corner of the letter and then drawing to the other "turning" points (shown by a • in the diagram of the E) of the letter in

succession. The coordinates of the points are given in DATA statements.
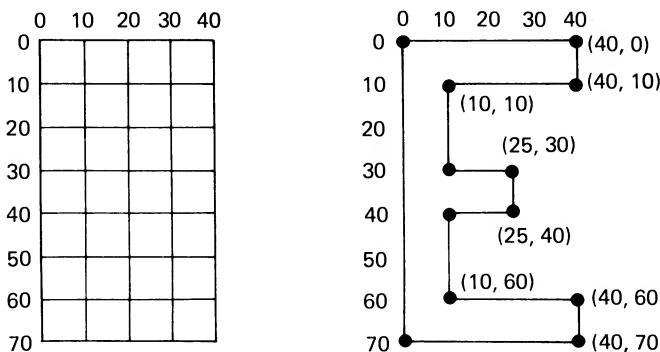
FIGURE 11.1   Grid Diagram of E

```
100 REM . Chapter 11, No. 2 (REMARK)
101 REM . J. Reisinger - NOV 82
102 REM . Describing variables with
103 REM . REMark statements
104 REM .........................
110 GRAPHICS 7+16
120 SETCOLOR 0,4,4:REM .COLOR 1=RED
130 SETCOLOR 1,0,10:REM COLOR 2=WHITE
140 SETCOLOR 2,9,4:REM .COLOR 3=BLUE
150 REM . PART   = Number of letter
160 REM . XSTART = X-coordinate of
161 REM .          upper left corner
162 REM .          of letter
170 REM . YSTART = Y-coordinate of
171 REM .          upper left corner
172 REM .          of letter
180 REM . COL    = Color number
190 REM . NUMPTS = Number of points
191 REM .          in letter
200 REM . POINT  = Loop counter for
201 REM .          reading coordi-
202 REM .          nates of points
210 FOR PART=1 TO 4
220 READ XSTART,YSTART
230 READ COL
240 READ NUMPTS
250 COLOR COL
260 PLOT XSTART,YSTART
270 FOR POINT=1 TO NUMPTS
280 READ X,Y
290 DRAWTO X+XSTART,Y+YSTART
300 NEXT POINT
310 DRAWTO XSTART,YSTART
320 NEXT PART
330 GOTO 330:REM . endless loop - end
499 REM .............................
500 REM . Data for outside of "R"
```

```
510 DATA 5,10,1,10
520 DATA 0,70,10,70,10,40,18,40,28,70
521 DATA 40,70,30,40,40,30,40,10,30,0
600 REM . Data for inside of "R"
610 DATA 15,20,1,5
620 DATA 0,20,15,20,20,15,20,5,15,0
700 REM . Data for "E"
710 DATA 55,10,2,11
720 DATA 0,70,40,70,40,60,10,60,10,40
721 DATA 25,40,25,30,10,30,10,10
722 DATA 40,10,40,0
800 REM . Data for "M"
810 DATA 105,10,3,11
820 DATA 0,70,10,70,10,22,25,49,40,22
821 DATA 40,70,50,70,50,0,40,0,25,27
822 DATA 10,0
999 REM ======== END OF LIST ========
```

Notice that in the last program we took care that no line had a length of more than 39 characters. This ensures that when the program is LISTed on the screen, no program line will overlap the screen line. This is sometimes handy for debugging and/or modifying your programs. On the other hand, if you have an 80-column printer at your disposal, you can make program lines of up to 80 characters and then work from the hard copy for debugging.

We have shown several variations of using REMark statements in programs. Aside from their informational function, they can give a certain visual structure to a program list so that the individual parts of the program are easier to find.

Line Numbers

When we talked about line numbers in Chapter 2, we indicated that we would not use line numbers less than 100 in programs. Our reason is purely one of style: If all of our line numbers are three-digit numbers, then the left margin of our program list stays straight. (We confess to being picky!) Normally, we number the lines at intervals of 10, but REMark statements may be given odd numbers to distinguish them from program statements. In the previous program, we numbered continuing REMark or DATA statements with consecutive numbers—this indicates that no other statement should be placed between these statements. Review the following lines:

```
 150 REM . PART    = Number of letter
 160 REM . XSTART  = X-coordinate of
 161 REM .           upper left corner
 162 REM .           of letter
```

```
 170 REM . YSTART = Y-coordinate of
 171 REM .              upper left corner
 172 REM .              of letter
 180 REM . COL     = Color number


 600 REM . Data for inside of "R"
 610 DATA 15,20,1,5
 620 DATA 0,20,15,20,20,15,20,5,15,0
 700 REM . Data for "E"
 710 DATA 55,10,2,11
 720 DATA 0,70,40,70,40,60,10,60,10,40
 721 DATA 25,40,25,30,10,30,10,10
 722 DATA 40,10,40,0
```

As you are working on a program, you may find it necessary to change the line numbers. This is not a simple task in ATARI BASIC. (Some versions of BASIC have a command to renumber the lines of a program automatically.) The following paragraphs give some suggested procedures.

Suppose that your program is short enough for all of it to be displayed on the screen at one time. Type in the following sample and follow the indicated steps.

Changing Line Numbers ("Renumbering")

```
 53 REM - Chapter 11, No. 3
 90 GRAPHICS 3
 95 COLOR 1
100 PLOT 0,0
121 DRAWTO 39,0
125 DRAWTO 39,19
137 DRAWTO 0,19
138 DRAWTO 0,0
```

To renumber the program at intervals of 10 starting with line 100, we'll relocate the program with line numbers in the 200s, and then move it back to the 100s. Follow these steps:

1. **Move the cursor to the beginning of line 53.**

2. **Type 200 in place of 53 and press the RETURN key.**

3. **Type 210 in place of 90 and press the RETURN key.**

4. **Continue line by line in this way, increasing the line number by 10 each time.**

5. **LIST the program. You should see this:**

```
 53 REM - Chapter 11, No. 3
 90 GRAPHICS 3
 95 COLOR 1
100 PLOT 0,0
```

```
121 DRAWTO 39,0
125 DRAWTO 39,19
137 DRAWTO 0,19
138 DRAWTO 0,0
200 REM - Chapter 11, No. 3
210 GRAPHICS 3
220 COLOR 1
230 PLOT 0,0
240 DRAWTO 39,0
250 DRAWTO 39,19
260 DRAWTO 0,19
270 DRAWTO 0,0
```

Now continue the procedure:

6. **Delete each line with a number less than 200 by typing the line number and then pressing the RETURN key.**

7. **A LIST of your program now looks like this:**

```
200 REM - Chapter 11, No. 3
210 GRAPHICS 3
220 COLOR 1
230 PLOT 0,0
240 DRAWTO 39,0
250 DRAWTO 39,19
260 DRAWTO 0,19
270 DRAWTO 0,0
```

Now, to change the numbers back to 100s instead of 200s:

8. **Move the cursor to the beginning of line 200, type 100, and press the RETURN key.**

9. **Type 110 instead of 210, and press the RETURN key.**

10. **Continue in this manner until all of the line numbers have been changed.**

11. **Delete all of the lines with numbers greater than or equal to 200.**

With luck, your LIST now looks like this:

```
100 REM - Chapter 11, No. 3
110 GRAPHICS 3
120 COLOR 1
130 PLOT 0,0
140 DRAWTO 39,0
150 DRAWTO 39,19
160 DRAWTO 0,19
170 DRAWTO 0,0
```

Whew! We said it wasn't simple!

The important principle in renumbering is to relocate your *entire program,* using line numbers larger than any in your original program. Otherwise, you'll certainly run the risk of interspersing old and new lines and accidentally deleting lines that you need.

If you have access to a disk drive, you can speed up the process somewhat in the following way:

1. **Relocate the program, following steps (1) to (5) as outlined in the preceding paragraphs.**

2. **Note the first line number (200 in our example) and the last line number (270 in our example) of the relocated program. Store the relocated program on a diskette with this command:**

```
LIST "D:NEWNUM.LST",200,270
```

   This command stores just line numbers 200 through 270 on the disk. The LIST command stores the program on the disk in a slightly different way than the SAVE command. We retrieve a LISTed program by giving an ENTER command instead of a LOAD command. To continue our procedure:

3. **Type NEW to erase the program from the computer memory. This is necessary because ENTERing a program doesn't erase the existing program; it adds the ENTERed one to it.**

4. **Retrieve the relocated program with the command** ENTER **"D:NEWNUM.LST".**

5. **Type LIST to verify that the relocated program is in the computer.**

6. **Repeat steps (8) to (10) as in the previous example; that is, change all of the 200 line numbers to 100 line numbers.**

If your program is too long for all of it to be displayed on the screen at one time, the procedure for renumbering it is essentially the same as the one outlined in the preceding paragraphs. It will be easier, though, if you have a printed copy of the program to work from, so that you can keep track of which lines of the program have been renumbered.

> ADVICE: It is always a good idea to have a copy
>   of your program on cassette or disk-
>   ette before you renumber it.

The following sample program (Chapter 11, No. 4) shows an example of poor screen display and an example of improved screen display. The poor example ENDs at line 170. To continue with

Improving
Screen Output

the improved example, type CONT and then press the RETURN key, or type GOTO 210 and then press the RETURN key.

```
100 REM - Chapter 11, No. 4
101 REM - Sloppy screen display
110 GRAPHICS 0
120 PRINT "    SLOPPY SCREEN DISPLAY"
130 PRINT "    -------------------"
140 PRINT "THIS PROGRAM DEMONSTRATES WHAT WILL PROBABLY"
150 PRINT "HAPPEN IF YOU DON'T PAY ATTENTION TO THE"
160 PRINT "LENGTH OF PRINT LINES."
170 END
200 REM - Better screen display
210 GRAPHICS 0:PRINT :PRINT
220 SETCOLOR 2,12,0
230 SETCOLOR 1,12,6
240 SETCOLOR 4,12,0
250 PRINT "        BETTER SCREEN DISPLAY"
260 PRINT "        -------------------"
270 PRINT :PRINT
280 PRINT "    This program demonstrates what":PRINT
290 PRINT "    a screen display can look like":PRINT
300 PRINT "    if you pay  attention  to  the":PRINT
310 PRINT "    length of print lines."
320 FOR K=1 TO 8:PRINT :NEXT K
```

Don't erase the program—you'll want to refer to it during the following discussion.

Good screen display is partly a matter of style, but here are some general principles that will make your programs look better:

1. Allow plenty of space on the screen for the printout. Leave a blank line or two at the top, and leave generous margins on the left and right. This makes the printout much easier to read. Run both programs again and compare the spacing.

2. Type most text using upper- and lowercase letters as in normal typing. Use capital letters for titles. Skip a line between lines of print if it makes the display easier to read. Center titles carefully, and insert extra spaces in print lines for blocked text, as in our example.

3. Use a colored background for variety. The SETCOLOR command determines the screen color in graphics modes 0, 1, and 2. Here's a review:

    ```
    SETCOLOR 2 - Screen background

    SETCOLOR 4 - Screen border

    SETCOLOR 1 - Luminance of text
    ```

    For the most pleasing contrast between the text and the background, make the difference between the luminance

numbers for the text and background equal to 6. The lumi-
nance number is the last number in the SETCOLOR state-
ment. For example, in our "improved" program, we used

```
SETCOLOR 2,12,0  for the background, and
SETCOLOR 1,12,6  for the text.
```

For dark letters on a light background, try

```
SETCOLOR 2,12,8  for the background, and
SETCOLOR 1,12,2  for the text.
```

Try making the border a different color from the back-
ground.

4. Use graphics modes 1 and 2 if you have a relatively small
amount of text to print. This is well worth the small extra
effort in programming that it takes. Here are versions of
our sample "improved" program in graphics modes 1 and 2:

```
100 REM - Chapter 11, No. 5
101 REM - Graphics 1 screen display
110 GRAPHICS 1
120 SETCOLOR 0,4,6:REM .. Upper case
130 SETCOLOR 1,12,6:REM . Lower case
140 SETCOLOR 2,12,0:REM . Background
150 SETCOLOR 4,12,0:REM . Border
160 POSITION 3,2
170 PRINT #6;"SCREEN DISPLAY"
180 POSITION 3,3
190 PRINT #6;"--------------"
200 POSITION 1,5
210 PRINT #6;"this program demon-"
220 POSITION 1,7
230 PRINT #6;"strates a  graphics"
240 POSITION 1,9
250 PRINT #6;"1 screen display in"
260 POSITION 1,11
270 PRINT #6;"place of  the usual"
280 POSITION 1,13
290 PRINT #6;"graphics 0 display."

100 REM - Chapter 11, No. 6
101 REM - Graphics 2 screen display
110 GRAPHICS 2
120 SETCOLOR 0,4,6:REM .. Upper case
130 SETCOLOR 1,12,6:REM . Lower case
140 SETCOLOR 2,12,0:REM . Background
150 SETCOLOR 4,12,0:REM . Border
160 POSITION 3,1
170 PRINT #6;"SCREEN DISPLAY"
```

```
180 POSITION 3,2
190 PRINT #6;"--------------"
200 POSITION 1,4
210 PRINT #6;"this program demon-"
220 POSITION 1,5
230 PRINT #6;"strates a  graphics"
240 POSITION 1,6
250 PRINT #6;"2 screen display in"
260 POSITION 1,7
270 PRINT #6;"place of  the usual"
280 POSITION 1,8
290 PRINT #6;"graphics 0 display."
```

5. Make the prompts for user input as clear as possible. Give examples if possible. The following programs show a confusing prompt and then an appropriate prompt.

```
100 REM - Chapter 11, No. 7
101 REM - WEAK PROMPTS
110 GRAPHICS 0:PRINT
120 PRINT "AREA of a TRIANGLE"
130 PRINT "------------------"
140 PRINT
150 PRINT "TYPE THE LENGTH AND WIDTH OF A"
160 PRINT "TRIANGLE AND THE COMPUTER WILL"
170 PRINT "GIVE YOU THE AREA."
180 INPUT LENGTH,WIDTH
190 AREA=LENGTH*WIDTH/2
200 PRINT
210 PRINT "AREA = ";AREA
```

```
100 REM - Chapter 11, No. 8
101 REM - BETTER PROMPTS
110 GRAPHICS 0:PRINT
120 PRINT "          AREA of a TRIANGLE"
130 PRINT "          ------------------"
140 PRINT
150 PRINT "   This program will give the area":PRINT
160 PRINT "   of a triangle if you type the":PRINT
170 PRINT "   length and width."
180 PRINT :PRINT
190 PRINT "        LENGTH ";:INPUT LENGTH
200 PRINT
210 PRINT "        WIDTH  ";:INPUT WIDTH
220 AREA=LENGTH*WIDTH/2
230 PRINT :PRINT
240 PRINT "       AREA = ";AREA
250 PRINT :PRINT :PRINT
```

6. Blink words or phrases in the screen display by printing and then erasing them repeatedly. This technique can be used to

emphasize key words or to indicate what quantity the user should input. Run the following program as an example:

```
100 REM - Chapter 11, No. 9
101 REM - Blinking word demo
110 GRAPHICS 2
120 POSITION 3,2
130 PRINT #6;"SOME OF THESE"
140 POSITION 4,4
150 PRINT #6;"WORDS BLINK"
160 POSITION 5,6
170 PRINT #6;"10 TIMES."
179 REM --------blink routine--------
180 FOR BLINK=1 TO 10
190 FOR WAIT=1 TO 100:NEXT WAIT
200 POSITION 10,4
209 REM - Replace word with spaces
210 PRINT #6;"      "
220 FOR WAIT=1 TO 50:NEXT WAIT
230 POSITION 10,4
239 REM - Replace spaces with word
240 PRINT #6;"BLINK"
250 NEXT BLINK
251 REM -----end of blink routine-----
```

The speed of the blinking can be adjusted by changing the length of the loops at lines 190 and 220.

7. Have someone check your screen output for spelling and grammatical errors. The user tends to lose confidence in your program if he or she encounters errors in English. It is not satisfactory for you to take the attitude, "Oh, they know what I mean." Do everything you can in your programming to help make this motto true: *Computers is almost perfect. They seldom make misteaks.*

EXERCISES

1. Complete the REMark statements in the following program so that they serve these three functions:

   a. Identify the program.

   b. Describe what a section of the program does.

   c. Describe what the variables are used for.

```
100 REM - Chapter 9, #19

101 REM

102 REM

110 DIM A$(4)

111 REM

120 A$="LOVE"
130 GRAPHICS 0:PRINT

131 REM

140 FOR A=1 TO 4

141 REM

150 FOR B=1 TO 4

151 REM

160 IF B=A THEN 270
170 FOR C=1 TO 4

171 REM

180 IF C=A THEN 260
190 IF C=B THEN 260
200 FOR D=1 TO 4

201 REM

210 IF D=A THEN 250
220 IF D=B THEN 250
230 IF D=C THEN 250

231 REM

240 PRINT A$(A,A);A$(B,B);A$(C,C);A$(D,D)
250 NEXT D
260 NEXT C
270 NEXT B
280 NEXT A
```

2. Run this short program to see how the screen display could be improved, and then rewrite it, using the guidelines in this chapter.

```
100 REM - Exercises 11, Problem 2.
110 GRAPHICS 0:PRINT
120 PRINT "THIS IS THE TITLE"
130 PRINT "THIS DISPLAY COULD BE IMPROVED IF A BIT MORE"
140 PRINT "CARE WERE TAKEN TO USE SPACE EFFECTIVELY."
150 PRINT
```

```
160 PRINT "NUMBER";"SQUARE"
170 FOR N=1 TO 10
180 PRINT N;" ";N*N
190 NEXT N
```

3. Rewrite this program so that the display is in GRAPHICS 2.

```
100 REM - Exercises  11, Problem 3.
110 GRAPHICS 0:PRINT
120 PRINT "AREAS OF TRIANGLES"
130 PRINT "-----------------"
140 PRINT
150 PRINT "BASE   HEIGHT   AREA"
160 PRINT "----   ------   ----"
170 FOR N=1 TO 3
180 READ BASE,HEIGHT
190 AREA=BASE*HEIGHT/2
200 PRINT " ";BASE;"        ";HEIGHT;"        ";AREA
210 PRINT
220 NEXT N
230 DATA 11,12,22,12,18,15
```

4. Modify this program so that the display has a dark green background with light green text.

```
100 REM - Exercises 11, Problem 4.
110 GRAPHICS 0
120 POKE 752,1
130 FOR TIMES=1 TO 100
140 X=INT(39*RND(0))
150 Y=INT(23*RND(0))
160 POSITION X,Y
170 PRINT #6;CHR$(36)
180 NEXT TIMES
190 GOTO 190
```

5. Modify the last program in this chapter (the "Blinking word demo" program) so that after the user types in his or her name, the letters will appear one-by-one, centered on the GRAPHICS 2 screen, each one blinking for about 3 seconds. After each letter has finished blinking, the next one is printed, so that at the end, the entire name is on the screen.

6. Type in the following program, and add appropriate prompts at lines 250 and 270. Run the program for several different inputs to be sure it is working properly. The prompts may occupy more than one line—the object of this exercise is to make them clear to the user. At line 190, type "high" in inverse video.

```
100 REM - Exercises 11, Problem 6.
110 GRAPHICS 2:PRINT
120 SETCOLOR 0,14,6
130 SETCOLOR 1,9,4
140 SETCOLOR 2,0,0
150 SETCOLOR 3,4,4
160 POSITION 3,3
170 PRINT #6;"--------------"
180 POSITION 5,4
190 PRINT #6;"high & low"
200 POSITION 4,6
210 PRINT #6;"TEMPERATURES"
220 POSITION 3,7
230 PRINT #6;"--------------"
240 FOR WAIT=1 TO 1000:NEXT WAIT
250 REM : Place a prompt at this line
260 INPUT CELSHIGH
270 REM : Place a prompt at this line
280 INPUT CELSLOW
290 FAHRHIGH=1.8*CELSHIGH+32
300 FAHRLOW=1.8*CELSLOW+32
310 GRAPHICS 0:PRINT :PRINT
320 PRINT "      FAHRENHEIT   CELSIUS"
330 PRINT "      ----------   -------"
340 PRINT
350 PRINT "HIGH    ",FAHRHIGH,CELSHIGH
360 PRINT
370 PRINT "LOW     ",FAHRLOW,CELSLOW
380 FOR SKIP=1 TO 12:PRINT :NEXT SKIP
```

7. Type in and run the following program exactly as it appears below, then make the suggested modifications.

```
100 REM - Exercises 11, Problem 7.
110 GRAPHICS 0:PRINT :PRINT :PRINT
120 PRINT "   THE OBJECT OF THIS PROGRAM IS TO"
130 PRINT "   SHOW THAT A MIXTURE OF UPPER AND"
140 PRINT "   LOWER CASE IS EASIER TO READ THAN"
150 PRINT "   UPPER CASE ALONE, AND THAT DOUBLE"
160 PRINT "   SPACING IS EVEN BETTER IF YOU HAVE"
170 PRINT "   ENOUGH SPACE ON THE SCREEN. "
180 PRINT :PRINT :PRINT
190 END
```

Modifications:

a. Renumber the program lines from 200 to 290 *without erasing* the original program. In lines 220 to 270, change the uppercase letters to lowercase, except for the first word of the sentence. Now run the modified program by typing

```
GOTO 200
```

and pressing the RETURN key. Compare the screen display with the original one for readability. You can look at the original display at any time by typing
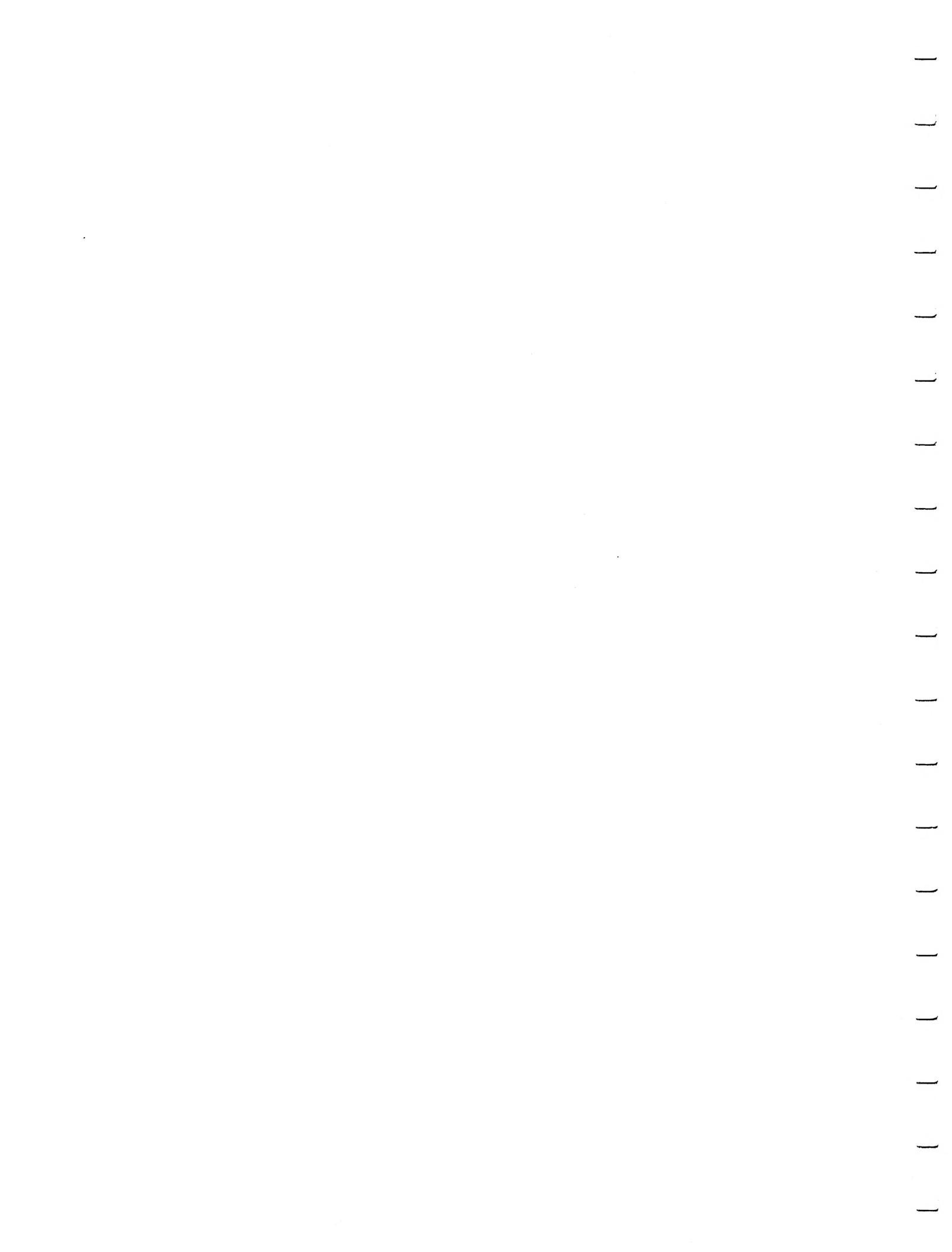
```
RUN
```

and pressing the RETURN key.

b. Add the following at the end of lines 220 through 260 to produce double-spacing:

```
:PRINT
```

Type

```
GOTO 200
```

and admire your results.

12

More About Graphics

In Chapter 3 we discussed graphics modes 0, 1, 2, 3, 5, and 7 at length, and we used these modes extensively in Chapters 4 and 10. There are also graphics modes 4, 6, 8, 9, 10, and 11, which we will discuss in this chapter. Modes 4 and 6 are similar to modes 5 and 7, but with fewer colors; mode 8 is the high resolution mode.

Modes 9, 10, and 11 work only with the GTIA television chip, which has been installed at the ATARI factory in all ATARI computers manufactured after January 1982. If your computer has the older CTIA television chip, the programs for graphics 9, 10, and 11 will either not run at all, or not show the intended colors and color changes.

In Chapter 3, we pointed out that in graphics mode 0 (the normal text mode), SETCOLOR 2 controls the color of the background and SETCOLOR 4 controls the color of the border. SETCOLOR 1 controls the luminance of the characters. The normal text mode uses a difference of 6 for the luminance numbers in the SET-COLOR 2 and SETCOLOR 1 statements. For example, if we use a pink background (SETCOLOR **2,4,10**), then a normal contrast between the characters and the background will be obtained by using SETCOLOR **1,4,4** (same color, lighter luminance: 10-4=6.) Run the following program for an illustration.

```
100 REM - Chapter 12, No. 1
101 REM - Graphics 0 Demo
110 DIM BACKCOLOR$(10),BORDCOLOR$(10)
120 FOR K=1 TO 6
125 GRAPHICS 0
```

```
130 READ BACKCOLOR$,BORDCOLOR$
140 READ BACKHUE,BACKLUM,BORDHUE,BORDLUM
150 SETCOLOR 2,BACKHUE,BACKLUM
160 SETCOLOR 4,BORDHUE,BORDLUM
162 IF BACKLUM<10 THEN SETCOLOR 1,0,BACKLUM+6:GOTO 170
164 IF BACKLUM>9 THEN SETCOLOR 1,0,BACKLUM-6
170 POSITION 10,5
180 PRINT "* ";BACKCOLOR$;" BACKGROUND"
182 POSITION 15,7
184 PRINT "SETCOLOR 2,";BACKHUE;",";BACKLUM
190 POSITION 10,10
200 PRINT "* ";BORDCOLOR$;" BORDER"
202 POSITION 15,12
204 PRINT "SETCOLOR 4,";BORDHUE;",";BORDLUM
210 FOR WAIT=1 TO 1000:NEXT WAIT
220 NEXT K
230 DATA WHITE,RED
232 DATA 0,14,4,4
240 DATA BLUE,YELLOW
242 DATA 7,2,14,10
250 DATA BLACK,GRAY
252 DATA 0,0,0,6
260 DATA PINK,VIOLET
262 DATA 4,10,5,2
270 DATA RUST,GOLD
272 DATA 2,0,1,6
280 DATA GREEN,GREEN
282 DATA 12,4,12,4
```

The DEFAULT VALUE for SETCOLOR 1 is

```
SETCOLOR 1,0,10
```

This means that if we use a luminance value of 10 in SETCOLOR 2 (for example, SETCOLOR 2,4,10), then the characters will be invisible on the background, because the difference between the luminance values in the SETCOLOR 1 and SETCOLOR 2 statements is 0 (10-10=0).

As an experiment, delete lines 162 and 164 from the previous program, and note how the text disappears when we have a pink background. This happens because the luminance for SET-COLOR 1 and SETCOLOR 2 are both equal to 10.

Graphics 8

Graphics 8 is the *high resolution* (HI-RES or HIRES) graphics mode. The screen is 320 pixels wide and 160 pixels high, but we pay a price—we can show only two colors on the screen at a time instead of four. The reason is simple—color graphics require a considerable amount of memory. Graphics mode 8 is similar to graphics mode 0—SETCOLOR 4 controls the color of the border, and SETCOLOR 2 controls the color of the background and

text window. SETCOLOR 1 controls the luminance of the graphics points. Try this example:

```
100 REM - Chapter 12, No. 2
101 REM - Graphics 8 Demo
110 GRAPHICS 8
120 SETCOLOR 4,4,4
130 SETCOLOR 2,0,10
140 SETCOLOR 1,0,4
150 COLOR 1
160 PLOT 125+D,30+E
170 FOR K=1 TO 9
180 READ X,Y:DRAWTO X+D,Y+E
190 NEXT K
200 D=D+3:E=E+4
210 IF D>=21 THEN 240
220 RESTORE
230 GOTO 160
240 PRINT ,"        GRAPHICS 8"
250 PRINT ,"         GRAPHICS 8"
260 PRINT ,"          GRAPHICS 8"
270 GOTO 270
300 DATA 125,30,195,30,195,70
310 DATA 120,90,120,130,200,130
320 DATA 200,90,125,70,125,30
```

Note that we use COLOR 1 with SETCOLOR 1. The color of the points plotted is the same as the background color, but they are either lighter or darker than the background. The critical numbers are the luminance values at the end of the SETCOLOR 1 and SETCOLOR 2 statements. For example, to produce a light graphics display on a dark background, change lines 130 and 140 as follows:

```
130 SETCOLOR 2,4,8
140 SETCOLOR 1,0,14
```

Now try minimal contrast:

```
130 SETCOLOR 2,0,0
140 SETCOLOR 1,0,2
```

Graphics 8 is particularly useful for graphing equations. Here is a program that will plot the graph of a linear equation of the form

$$Ax + By = C,$$

where the user inputs the values for A, B, and C.

```
100 REM - Chapter 12, No. 3
101 REM - Plot of linear equation
110 GRAPHICS 2
115 SETCOLOR 0,4,4:SETCOLOR 1,14,10
120 POSITION 5,3
```

```
130 PRINT #6;"    PLOT"
140 POSITION 5,5
150 PRINT #6;"aX + bY = c"
160 PRINT "Type:    A,B,C    then press RETURN."
165 PRINT "                      (Example:   2,5,10)"
170 PRINT "A,B,C ";:INPUT A,B,C
175 IF A=0 AND B=0 THEN ? CHR$(253);"OOPS!":GOTO 160
180 FOR WAIT=1 TO 500:NEXT WAIT
190 GRAPHICS 8
200 SETCOLOR 4,4,4
210 SETCOLOR 2,0,10
220 SETCOLOR 1,0,4
230 COLOR 1
240 REM *** PLOT AXES ***
250 PLOT 0,80:DRAWTO 319,80
260 PLOT 160,0:DRAWTO 160,155
270 REM *** PLOT TICK MARKS ON AXES ***
280 FOR X=10 TO 310 STEP 10
290 PLOT X,78:DRAWTO X,82
300 NEXT X
310 FOR Y=10 TO 150 STEP 10
320 PLOT 158,Y:DRAWTO 162,Y
330 NEXT Y
332 IF B<>0 THEN 340:REM *** TEST FOR VERTICAL LINE ***
334 PLOT 160+10*C/A,0:DRAWTO 160+10*C/A,155:GOTO 390
340 REM *** PLOTTING ROUTINE ***
350 FOR X=-15 TO 15 STEP 0.1
355 TRAP 375:REM - AVOID ERROR 141 -
360 Y=(C-A*X)/B
370 PLOT 160+10*X,80-10*Y:GOTO 380
375 TRAP 40000
380 NEXT X
390 ? :? "        GRAPH OF   ";A;"x + ";B;"y = ";C
```

Let's consider a few aspects of the program:

1. **At lines 200, 210, and 220, we set the colors for the border (SETCOLOR 4), the background (SETCOLOR 2), and the luminance of the points being plotted (SETCOLOR 1). Note that the *difference* between the *luminance* values in the SETCOLOR 2 statement and the SETCOLOR 1 statement is 6. This provides a good contrast between the graph and the background.**

2. **If B=0, then the equation**

$$Ax + By = C$$

reduces to

$$x = C/A$$

which is the equation of a vertical line. This test is made at line 332, then the vertical line is plotted at line 334. In this case

we don't need to calculate any intermediate values, so we can skip the plotting routine and go to line 390.

3. In lines 270 to 330, we drew the "tick marks" on the axes at intervals of 10 pixels. That is, 10 pixels will equal one unit on our graph. Our graph will extend from -15 to 15 on the X-axis. This is indicated in line 350. Note that the step is .1 (one-tenth) because each unit is represented by 10 pixels.

   At line 360 we calculate the Y-value that corresponds to each X-value (Could you solve the equation Ax + By = C for Y?)

   Because the origin of our graph is at the point 160,80 (the middle of the graphics 8 screen), we add ten times the X-value to 160, and subtract ten times the Y-value from 80 at line 370. (We *subtract* the Y-value because the Y-coordinates on the screen *increase* from top to bottom, while on a mathematical graph, they *decrease* from top to bottom.)

4. The TRAP statement at line 355 sends control to line 375 if an error occurs. If the value 80–10*Y is greater than 159 or less than zero, error 141 would stop the program. Instead, the TRAP is reset (so it can be used again) at line 375 and then the next X-value is processed. To reset the TRAP, we always use a line number greater than 32767 (the greatest line number that we can use in ATARI BASIC).

We could plot other equations by changing the equation at line 360. We could also change the scale of the axes. Just for fun, delete line 390 and change line 360 to:

```
360 Y=5*SIN(x/2)
```

When you run the program, type in **1,1,1** for **A,B,C**. You'll see a graph of the sine function.

GRAPHICS 4 and GRAPHICS 6

GRAPHICS 4 and GRAPHICS 6 have the same size screens as GRAPHICS 5 and GRAPHICS 7, respectively, but we can plot points in just one color instead of three. This might be an advantage if we want to save memory because GRAPHICS 4 and 6 use only about one-half as much memory as GRAPHICS 5 and 7.

Try these two short sample programs:

```
100 REM - Chapter 12, No. 4
101 REM - Graphics 4 Demo
110 GRAPHICS 4
120 COLOR 1
130 PLOT 23,3
140 FOR K=1 TO 14
```

```
150 READ X,Y
160 DRAWTO X,Y
170 NEXT K
300 ? :? "            THIS IS GRAPHICS 4"
310 DATA 23,28,43,28,43,35,53,35
320 DATA 53,28,63,28,63,20,53,20
330 DATA 53,10,43,10,43,20,33,20
340 DATA 33,3,23,3

100 REM - Chapter 12, No. 5
101 REM - Graphics 6 Demo
110 GRAPHICS 6
120 COLOR 1
130 PLOT 20,15
140 FOR K=1 TO 12
150 READ X,Y
160 DRAWTO X,Y
170 NEXT K
180 PLOT 45,25:DRAWTO 67,47
182 DRAWTO 89,25:DRAWTO 45,25
190 PLOT 104,25:DRAWTO 74,55
192 DRAWTO 117,55:DRAWTO 117,25
194 DRAWTO 105,25
300 ? :? "            THIS IS GRAPHICS 6"
310 DATA 20,25,30,25,60,55,20,55
320 DATA 20,65,140,65,140,55,130,55
330 DATA 130,25,140,25,140,15,20,15
```

Table 12-1 summarizes the use of the SETCOLOR and COLOR commands in GRAPHICS 4 and GRAPHICS 6:

TABLE 12-1   SETCOLOR and COLOR Numbers—GRAPHICS 4 and 6

Default Color	SETCOLOR No.	COLOR No.	Feature Controlled by the SETCOLOR or COLOR Command
orange	0	1 or 3	Color of point to be plotted
blue	2	-	Color of the text window
black	4	-	Color of the background
	(4)	0 or 2 or 4	Color of point to be plotted. Because this is the background color, the point will be erased

Filling an Area With Color

Quite often in doing graphics work, we need to produce a solid area of color rather than just an outline figure. This can be done by plotting all the points in the interior of the figure in the desired color, but this might be extremely tedious. We can use a special

case of the XIO command for this purpose. Run the following program as an illustration. Study the REMark statements to see how the XIO command "fills" the flag.

```
100 REM - Chapter 12, No. 6
101 REM - Fill Demo
110 DIM F$(1)
120 GRAPHICS 5
130 COLOR 1
140 REM *** DRAW FLAG ***
150 PLOT 10,4:DRAWTO 69,4
160 DRAWTO 53,20:DRAWTO 69,36
170 DRAWTO 10,36
180 REM *** DRAW DIAGONALS HEAVY ***
190 PLOT 68,4:DRAWTO 52,20:DRAWTO 68,36
200 REM *** DRAW FLAGPOLE ***
210 FOR X=8 TO 10
220 PLOT X,2:DRAWTO X,39
230 NEXT X
240 ? "Press  RETURN  to see the fill."
250 INPUT F$
260 PLOT 10,36:REM *** BOTTOM OF FILL ***
270 POSITION 10,5:REM *** TOP OF FILL ***
280 POKE 765,3:REM *** COLOR OF FILL ***
290 XIO 18,#6,0,0,"S:":REM *** DO THE FILL ***
300 PRINT "        THAT WAS THE FILL."
310 FOR WAIT=1 TO 500:NEXT WAIT
320 PRINT :PRINT :PRINT
330 REM *** PLOT MESSAGE ***
340 COLOR 0
350 PLOT 20,25:DRAWTO 20,15:DRAWTO 25,15
360 PLOT 21,19:DRAWTO 24,19
370 PLOT 27,15:DRAWTO 27,25
380 PLOT 30,15:DRAWTO 30,25:DRAWTO 35,25
390 PLOT 37,15:DRAWTO 37,25:DRAWTO 42,25
```

Can you see what happened? Let's review the essential steps:

1. **Draw the top, right side, and bottom of the figure to be filled. We did this at lines 150 to 170.**

2. **PLOT the bottom left corner of the figure (line 260).**

3. **POSITION the graphics cursor at the top left corner of the figure (line 270).**

4. **POKE the number of the color for the area to be filled into memory location 765 (line 280).**

5. **Give the "fill" command (line 290). XIO is a general input/output (IO) command that can be used for several advanced**

programming techniques. To use it as a fill command, the form
is always the same:

```
XIO 18,#6,0,0,"S:"
```

Notice that the fill is accomplished by drawing *horizontal*
lines to the right from the left side of the figure. This side of the
figure is drawn by the XIO command, using the data given in
steps (2) and (3). Each horizontal line stops when it reaches a
colored pixel to the right. If there is no colored pixel to the right,
then the horizontal line wraps around; that is, it continues to the
right edge of the screen and then continues from the left edge of
the screen to the first colored pixel.

The fill can proceed either from bottom to top, as in our
example, or from top to bottom. To do the latter, change lines
260 and 270 as follows:

```
260 PLOT 10,4
270 POSITION 10,36
```

Do you see that the fill always proceeds from left to right? There
is no other possibility!

The XIO command draws the left side of the figure in what-
ever color has been indicated prior to steps (2) and (3). Thus we
can draw the left side in the same color as the fill, in a different
color, or even in the (invisible) background color.

Because fills work along straight lines, it is sometimes
necessary to fill a given figure by dividing it into two or more
parts. As an example, let's look again at the "Graphics 4 Demo
Chapter 12, No. 4," where we drew the outline of a figure 4. Let's
color the 4 orange. Run the original program again:

```
100 REM - Chapter 12, No. 7
101 REM - (Chapter 12, No. 4 - Graphics 4 Demo)
110 GRAPHICS 4
120 COLOR 1
130 PLOT 23,3
140 FOR K=1 TO 14
150 READ X,Y
160 DRAWTO X,Y
170 NEXT K
300 ? :? "            THIS IS GRAPHICS 4"
310 DATA 23,28,43,28,43,35,53,35
320 DATA 53,28,63,28,63,20,53,20
330 DATA 53,10,43,10,43,20,33,20
340 DATA 33,3,23,3
```

Now add the following lines:

```
101 REM - 1st addition to Chapter 12, No. 7
180 PLOT 43,35:POSITION 43,10
190 POKE 765,1
200 XIO 18,#6,0,0,"S:"
```

Can you write the lines to fill the other part of the 4? If you have trouble, here's some help:

```
101 REM - 2nd addition to Chapter 12, No. 7
210 PLOT 23,28:POSITION 23,3
220 POKE 765,1
230 XIO 18,#6,0,0,"S:"
```

While we're at it, let's make the color of the 4 alternate between yellow and red to produce a blinking effect. Add these lines:

```
101 REM - 3rd addition to Chapter 12, No. 7
240 FOR BLINK=1 TO 10
250 SETCOLOR 0,14,10
260 FOR WAIT=1 TO 100:NEXT WAIT
270 SETCOLOR 0,4,4
280 FOR WAIT=1 TO 100:NEXT WAIT
290 NEXT BLINK
```

Now, if you feel ready for a challenge, see if you can modify the "Graphics 6 Demo" to fill the Roman numeral VI symbol. If you succeed, then you have a good grasp of how to use the XIO command to fill an area!

This would be a good opportunity to experiment with some short programs of your own before we continue to GRAPHICS 9, 10, and 11. Have fun!

GRAPHICS 9

GRAPHICS 9, 10, and 11 have two things in common: (1) they all have a screen that is 80 pixels wide and 192 pixels high, and (2) they do not have text windows.

GRAPHICS 9 has sixteen luminances of the same color. The following program draws a picture with fifteen different color luminances of gray on a black background (the sixteenth luminance) and then changes to the other fifteen basic ATARI colors.

```
100 REM - Chapter 12, No. 8
101 REM - Graphics 9 Demo
110 GRAPHICS 9
120 FOR X=79 TO 9 STEP -5
130 C=16-(X-4)/5:REM *** C varies from 1 to 15 ***
140 COLOR C
150 FOR Y=100-X TO 191-2*X
160 PLOT X-6,Y:DRAWTO X,Y+6
170 NEXT Y
180 NEXT X
190 FOR W=1 TO 500:NEXT W
200 FOR S=1 TO 15
210 SETCOLOR 4,S,0
220 FOR W=1 TO 500:NEXT W
230 NEXT S
240 GOTO 240
```

You can see that GRAPHICS 9 is useful to suggest three-dimensional graphics displays. Note, though, that the pixels are long and narrow, rather than almost square as in the other graphics modes. The following short program illustrates this.

```
100 REM - Chapter 12, No. 9
101 REM - Graphics 9 pixel Demo
110 GRAPHICS 9
120 SETCOLOR 4,14,0
130 FOR X=0 TO 79 STEP 2
140 FOR Y=0 TO 191 STEP 2
150 C=C+1:IF C=16 THEN C=1
160 COLOR C
170 PLOT X,Y
180 NEXT Y
190 NEXT X
200 GOTO 200
```

The program plots the pixels in every other row and column and in each of the fifteen luminances that are brighter than the background. Even though the pixels are all the same size, the brighter ones appear larger and closer to us.

In GRAPHICS 9, we use a statement of the form

```
SETCOLOR 4,C,0
```

to determine the color of the display. C can be any integer from 0 to 15, inclusive. Table 12-2 is a list of the colors that will be produced by the various values of C.

TABLE 12-2   SETCOLOR Hue Numbers

Value of C	Color Produced
0	Gray
1	Light orange (Gold)
2	Orange
3	Red-orange
4	Pink
5	Purple
6	Purple-blue
7	Blue
8	Blue
9	Light blue
10	Turquoise
11	Green-blue
12	Green
13	Yellow-green
14	Orange-green
15	Light orange

The *luminance* of the display in GRAPHICS 9 is determined by a statement of the form

```
COLOR L
```

where L is an integer from 1 to 15, inclusive. The larger the value of L, the brighter the luminance. Once points have been plotted, their luminance can be changed only by giving a new COLOR statement and replotting the points.

We can also use the fill procedure in GRAPHICS 9, as the following program shows. Study the REMark statements to review the use of the SETCOLOR and COLOR statements.

```
100 REM - Chapter 12, No. 10
101 REM - Graphics 9 fill Demo
110 GRAPHICS 9
120 REM - Set background color to green (12),
130 REM - with darkest luminance (0)
140 SETCOLOR 4,12,0
150 REM - Draw and fill a rectangle in each of
160 REM - the 15 other luminances (COLOR statement).
170 FOR LUM=1 TO 15
180 COLOR LUM:REM - Luminance of perimeter
190 PLOT 10,10*LUM:DRAWTO 70,10*LUM
200 DRAWTO 70,10*LUM+9:DRAWTO 10,10*LUM+9
210 POSITION 10,10*LUM
220 POKE 765,LUM:REM - Luminance of interior
230 XIO 18,#6,0,0,"S:"
240 NEXT LUM
250 GOTO 250:REM - Prevent display from disappearing
```

GRAPHICS 10

And now, nine colors on the screen at once! GRAPHICS 10 has nine colors in whatever luminances we choose. Do you remember the four paint pots from Chapter 3? These are the *color registers* (memory locations) where color information is stored. We're used to placing information in these registers with the SETCOLOR command, but an easier way to work with the nine color registers of GRAPHICS 10 is to use the POKE command.

Because each color register can hold only a single number, we need to convert the hue and luminance information for a particular color into a single number. The formula is simple: Multiply the hue number by 16 and add the luminance number. For example, pink has the hue number 4 and the luminance number 10, so we would use the number 74 (4 times 16 plus 10). The command to put this number into the color register at memory location 704 would be

```
POKE 704,74
```

Table 12-3 is an extension of Table 3-3 that converts the hue and luminance numbers into single numbers:

TABLE 12-3	GRAPHICS 10 Color Values		
Color	*Hue*	*Lum*	*16×Hue+Lum*
Black	0	0	0
White	0	14	14
Gray	0	6	6
Brown	14	0	224
Red	4	4	68
Orange	2	6	38
Yellow	14	10	234
Green	12	4	196
Blue	7	2	114
Violet	5	2	82
Pink	4	10	74
Flesh	3	12	60
Gold	1	6	22
Rust	2	0	32
Turquoise	10	8	168
Forest green	11	2	178
Olive drab	13	2	210
Screen blue	9	4	148

Now let's do a GRAPHICS 10 program that will draw eight colored stripes on a black background. The nine color registers for GRAPHICS 10 are located at memory locations 704 through 712. Location 704 holds the background color (default: black).

We'll change it to gray at line 120. Our stripes will have these colors: white, red, blue, yellow, green, orange, brown, and pink. COLOR 1 refers to register 705; COLOR 2 refers to register 706; COLOR 3, register 707, etc.

```
100 REM - CHAPTER 12, NO. 11
101 REM - Graphics 10 Demo
110 GRAPHICS 10
120 POKE 704,4:REM - Gray background
130 REM - Load color registers
140 FOR REG=705 TO 712
150 READ COLNUMBER
160 POKE REG,COLNUMBER
170 NEXT REG
180 DATA 14,68,114,234,196,38,226,74
190 REM - Draw outline of stripe
200 FOR C=1 TO 8
210 COLOR 1:REM - White for border of stripe
220 PLOT 10,20*C:DRAWTO 70,20*C
230 DRAWTO 70,20*C+19:DRAWTO 10,20*C+19
240 POSITION 10,20*C
250 POKE 765,C
260 XIO 18,#6,0,0,"S:":REM - Fill stripe
270 NEXT C
280 GOTO 280
```

Try changing the colors of the stripes by changing the numbers in line 180. Use Table 12-3. You can also change the color of the borders of the stripes by changing the color number at line 210; try COLOR C.

If we "rotate" the color values in the color registers in GRAPHICS 10, we can create the illusion of motion. Add the following lines to your current program:

```
100 REM - Chapter 12, No. 12
101 REM - Addition to Chapter 12, No. 11
102 REM - Graphics 10 color rotation
120 POKE 704,0:REM - Black background
180 DATA 226,228,230,232,194,196,198,200
210 COLOR C
215 REM - Stripes are staggered
220 PLOT 5*C,20*C:DRAWTO 5*C+35,20*C
230 DRAWTO 5*C+35,20*C+19:DRAWTO 5*C,20*C+19
240 POSITION 5*C,20*C
280 REM - Rotate colors
290 T=PEEK(705)
300 FOR K=705 TO 711
310 POKE K,PEEK(K+1)
320 NEXT K
330 POKE 712,T
```

```
340 FOR W=1 TO 30:NEXT W
350 GOTO 290
```

Do you see how the rotation works? Let's look at that part of the program in detail:

```
280 REM - Rotate colors
290 T=PEEK(705)
300 FOR K=705 TO 711
310 POKE K,PEEK(K+1)
320 NEXT K
330 POKE 712,T
340 FOR W=1 TO 30:NEXT W
350 GOTO 290
```

Immediately we see a new command: T=PEEK(705). We know that POKE places a value directly into a specified memory location. The PEEK command "reads" that value *without removing it*. The command T=PEEK(705) places the value stored at memory location 705 in variable T. Notice that the memory location we are PEEKing at is always placed in parentheses.

Now consider what happens at lines 300 to 320: In each of the memory locations 705 to 711 (the locations of seven of the color registers), we place the color value that was stored in the *next* color register. For example, if K=707, then line 310 is

```
310 POKE 707,PEEK(708)
```

which means that the contents of location 708 are transferred to location 707. But this means that the areas that had the color associated with register 707 now have the color associated with register 708. We do this for all of the locations 705 to 711 and place the contents of location 705 (which we stored in T in the beginning) in location 712 (line 330). Each of our eight colors has been rotated once. We pause a bit at line 340 and then rotate again. The result: apparent motion. You can speed it up or slow it down by changing the length of the loop at line 340.

Here is another example of rotating the colors in GRAPHICS 10 to produce apparent motion. We'll explain some details of the program after you've had a chance to run it.

```
100 REM - Chapter 12, No. 13
101 REM - Graphics 10 "wheel"
110 GRAPHICS 10
120 POKE 704,0
130 FOR REG=705 TO 712
```

```
140 READ C
150 POKE REG,C
160 NEXT REG
170 DATA 86,86,86,68,68,82,82,82
180 FOR X=5 TO 75
190 Q=INT(X/8):RE=1+X-Q*8
200 COLOR RE
210 X1=X-40
220 Y1=INT(SQR(900-900*X1*X1/1225)+0.5)
225 IF Y1=0 THEN Y2=0:GOTO 240
230 Y2=-Y1
240 PLOT X,40-Y2:DRAWTO X,150-Y2
250 COLOR 9-RE:PLOT X,37-Y1:DRAWTO X,37-Y2
260 NEXT X
270 REM - Rotate colors
280 T=PEEK(705)
290 FOR K=705 TO 711
300 POKE K,PEEK(K+1)
310 NEXT K
320 POKE 712,T
330 GOTO 280
340 GOTO 340
```

The mathematical idea of the program is to calculate the coordinates of the points on an ellipse using the equation

$$\frac{X^2}{1225} + \frac{Y^2}{900} = 1$$

which is equivalent to the two equations

$$Y_1 = \sqrt{900 - \frac{900X^2}{1225}}$$

and

$$Y_2 = -Y_1$$

The wheel involves two ellipses: one for the top, and one for the bottom. The upper ellipse has a center at **40,40;** and the lower one has a center at **40,150.** Y1 is the Y-coordinate on the top half of the ellipse, and Y2 is the Y-coordinate on the bottom half of the ellipse. To draw the front of the wheel, we simply connect the bottom of the upper ellipse with the bottom of the lower ellipse (line 240). The inside of the wheel is actually just the upper ellipse. However, to get it to "rotate" in the opposite direction from the front, we change the color value from RE to 9–RE (line 250). RE is one more than the remainder obtained by dividing the X-value by eight. This means that we have a repeat of the colors every eight vertical lines. Note that the data at line

170 provides for three light violet stripes, two red stripes, and then three darker violet stripes. This ellipse crosses the X-axis at 35 and -35, and it crosses the Y-axis at 30 and -30. Thus the long axis of the ellipse is 70 units long, and the short axis of the ellipse is 60 units long. Our graphics 10 screen is 80 pixels wide, so to get a length of 70, we start at X=5 and go to X=75 (line 180). But the X-values 5 and 75 really correspond to -35 and 35 on a graph, so we simply subtract 40 from the X-value (line 210: 5-40=-35; 75-40=35). Then we compute the corresponding Y-value at line 220.

The wheel actually involves two ellipses: one for the top, and one for the bottom. The upper ellipse has a screen center at 40,40; and the lower one has a screen center at 40,150. The centers are determined by lines 210 and 240. Do you remember why we subtract the Y-value of the graph from the Y-coordinate of the center of the screen? If not, refer to the explanation for program Chapter 12, No. 3.

GRAPHICS 11

Mode 11 is the reverse of mode 9: instead of one color and sixteen luminances, we have one luminance and sixteen colors. We can control the luminance but not the individual colors, as we could in GRAPHICS 10. Try this short sample program:

```
100 REM - Chapter 12, No. 14
101 REM - Graphics Mode 11 Demo
110 GRAPHICS 11
120 REM - Black background
130 REM - Darkest luminance
140 SETCOLOR 4,0,0
150 REM - Draw 16 vertical stripes
160 FOR HUE=0 TO 15
170 COLOR HUE
180 FOR X=0 TO 4
190 PLOT 5*HUE+X,0:DRAWTO 5*HUE+X,191
200 NEXT X
210 NEXT HUE
220 REM - Change luminance
230 FOR LUM=0 TO 14 STEP 2
240 SETCOLOR 4,0,LUM
250 FOR WAIT=1 TO 200:NEXT WAIT
260 NEXT LUM
270 GOTO 230:REM - Repeat luminance cycle
```

In GRAPHICS 11, the color of the pixel being plotted is determined by the COLOR statement (line 170 in the program above). If you run the program again, you will see that the colors

are almost, but not quite, arranged in rainbow order. The following program changes the order slightly to draw a rainbow.

```
100 REM - Chapter 12, No. 15
101 REM - Graphics 11 Blinking Rainbow
110 GRAPHICS 11
120 REM - Black background
130 LUM=4
140 SETCOLOR 4,0,LUM
150 REM - Draw 30 horizontal stripes
160 FOR K=1 TO 30
170 READ HUE
180 COLOR HUE
190 FOR Y=0 TO 5
200 PLOT 2,6*K+Y:DRAWTO 79,6*K+Y
210 NEXT Y
220 NEXT K
230 REM - Alternate luminance number
240 REM - between 4 and 6.  (Line 260)
250 FOR W=1 TO 30:NEXT W
260 LUM=10-LUM
270 SETCOLOR 4,0,LUM
280 GOTO 250
290 DATA 5,4,3,2,15,1,14,13,12,11,10,9,8,7,6
300 DATA 5,4,3,2,15,1,14,13,12,11,10,9,8,7,6
```

In GRAPHICS 11, the background color is determined by the middle number in the SETCOLOR statement. In the previous two programs, the line

```
140 SETCOLOR 4,0,0
```

produced a *black* background. The line

```
140 SETCOLOR 4,7,0
```

would have produced a *blue* background, and the line

```
140 SETCOLOR 4,12,0
```

would have produced a *green* background. However, background colors other than black may change the other colors used in the program. You'll have to experiment!

The third number in the SETCOLOR statement determines the base luminance. It is best to set this to zero; otherwise the number of available colors will be limited.

Here is one last sample program in GRAPHICS 11. Note that data lines 308, 309, 310, 311, 312, and 313 are the same as lines 306, 305, 304, 303, 302, and 301, respectively. You can duplicate a line by moving the cursor to its line number, typing the new line number, and then pressing the RETURN Key.

```
100 REM - Chapter 12, No. 16
101 REM - Graphics 11 Mosaic Picture
110 GRAPHICS 11
120 SETCOLOR 4,0,4
130 REM - Fill screen with colored squares
140 FOR R=2 TO 170 STEP 14
150 FOR C=11 TO 66 STEP 5
160 READ HUE
170 COLOR HUE
180 FOR L=0 TO 10
190 PLOT C,R+L:DRAWTO C+3,R+L
200 NEXT L
210 NEXT C
220 NEXT R
230 GOTO 230
301 DATA 9,10,10,9,8,5,4,3,2,15,14,13
302 DATA 10,9,8,5,4,3,2,15,14,13,12,11
303 DATA 9,8,5,4,3,2,15,14,13,12,11,10
304 DATA 8,5,4,3,2,15,14,13,12,11,10,9
305 DATA 8,5,4,3,2,15,14,13,12,11,10,9
306 DATA 5,4,3,2,15,14,13,12,11,10,9,5
307 DATA 5,4,3,2,15,14,13,12,11,10,5,4
308 DATA 5,4,3,2,15,14,13,12,11,10,9,5
309 DATA 8,5,4,3,2,15,14,13,12,11,10,9
310 DATA 8,5,4,3,2,15,14,13,12,11,10,9
311 DATA 9,8,5,4,3,2,15,14,13,12,11,10
312 DATA 10,9,8,5,4,3,2,15,14,13,12,11
313 DATA 9,10,10,9,8,5,4,3,2,15,14,13
```

EXERCISES

1. Fill in the hue and luminance numbers in each example to produce the given colors in GRAPHICS 0:

   a. Blue letters on a pale blue background with a red border.

SETCOLOR 1 , _____ , _____

SETCOLOR 2 , _____ , _____

SETCOLOR 4 , _____ , _____

b. Red letters on a pink background with a yellow border.

SETCOLOR 1 , _____ , _____

SETCOLOR 2 , _____ , _____

SETCOLOR 4 , _____ , _____

c. Gray letters on a black background with a white border.

SETCOLOR 1 , _____ , _____

SETCOLOR 2 , _____ , _____

SETCOLOR 4 , _____ , _____

d. Dark turquoise letters on a pale turquoise background with a brown border.

SETCOLOR 1 , _____ , _____

SETCOLOR 2 , _____ , _____

SETCOLOR 4 , _____ , _____

2. In GRAPHICS 8, there are _____ graphics pixels on the screen.

3. In GRAPHICS 8+16 (no text window), the screen is _____ pixels high.

4. The GRAPHICS 8+16 screen is exactly _____ times as big as the GRAPHICS 0 screen.
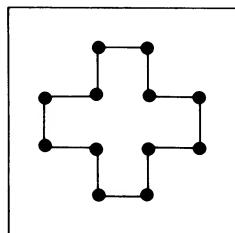
5. Does the fill command work in GRAPHICS 8? _____

6. Graphics modes 4 and 6 are similar to graphics modes 5 and 7.

    a. What is the advantage of using graphics modes 4 and 6?

    _____

    b. What is the disadvantage of using graphics modes 4 and 6?

    _____

7. Write a program in GRAPHICS 7 that will draw a yellow triangle on a gray background. Let the vertices of the triangle be the points **(80,10)**, **(140,70)** and **(20,70)**. Use the fill command to fill the triangle in yellow.

8. Write a program in GRAPHICS 3+16 that uses the fill command to draw a Swiss flag. The Swiss flag is a white cross formed from 5 squares on a red background. You will need to use the fill command twice. Here is the form of the flag:

9. Write a program in GRAPHICS 7+16 that uses the fill command to draw a traffic "STOP" sign. Recall that a "STOP" sign is a regular octagon.

10. Write a program in GRAPHICS 7+16 that will draw your initials in block letters (similar to the GRAPHICS 4 demo) and fill them with the color of your choice.

11. Graphics modes 9, 10, and 11 have _____ graphics pixels on the screen.

12. Do graphics modes 9, 10, and 11 have text windows? _____

13. What is the difference in size between GRAPHICS 9 (or 10 or 11) pixels and GRAPHICS 8 pixels?

_____

_____

14. Add the following lines to the program "Graphics 9 fill demo" to produce an interesting effect.

```
200 COLOR 10
210 PLOT 21,50:DRAWTO 60,50
220 DRAWTO TO 60,140:DRAWTO 21,140
230 POSITION 21,50
240 POKE 765,10
250 XIO 18,#6,0,0,"S:"
260 GOTO 260
```

Try replacing the number 10 at lines 200 and 240 with different values. (1=dark, 15=light.)

15. Modify the program "Graphics 9 fill demo" so that the rectangles are vertical instead of horizontal.

16. Modify the program "Graphics 10 demo" so that the background is black, the outline of the rectangles is the same color as the rectangle, and the eight rectangles have the following colors: gray, white, red, orange, pink, green, turquoise, blue.

17. Modify the program "Graphics 10 color rotation" so that the "steps" seem be moving downward instead of upward.

18. If you were successful with problem 17, modify it to produce two staircases—one moving upward, and the other moving downward.

19. Run the following GRAPHICS 10 program, which draws the ATARI symbol in rotating colors:

```
100 REM - Exercises 12, Problem 19.
110 GRAPHICS 10
120 FOR REG=705 TO 712
```

```
130 READ COLVAL
140 POKE REG,COLVAL
150 NEXT REG
160 DATA 68,72,38,234,196,200,116,120
170 COLR=1:Y=1
180 FOR X=10 TO 69
190 COLOR COLR
200 IF Y>141 THEN 300
210 PLOT X,141-Y:DRAWTO X,191-Y
220 PLOT 79-X,141-Y:DRAWTO 79-X,191-Y
230 FOR Q=36 TO 43
240 PLOT Q,191-Y:DRAWTO Q,141-Y
250 NEXT Q
260 Y=Y*1.23
270 COLR=COLR+1:IF COLR>8 THEN COLR=1
280 NEXT X
290 REM - Rotate colors
300 T=PEEK(705)
310 FOR K=705 TO 711
320 POKE K,PEEK(K+1)
330 NEXT K
340 POKE 712,T
350 GOTO 300
```

Modify the program so that the display is drawn in shades of gray on a white background.

20. Modify the DATA statements in the "Graphics 11 Mosaic Picture" to produce your own mosaic picture. Let your fantasy and imagination be your guide!

There are 12 numbers in each DATA line—these are the color value numbers for the "tiles" in a particular row of the screen display. You can change the number of tiles from thirteen rows of twelve each by changing the values in the three loops (variables R, C, and L) of the program.

13

Numeric Functions

So far, we have not really concentrated on the ATARI as a calculating device; that is, we have not emphasized its "number crunching" capabilities. To be sure, we have produced some tables of squares of numbers, etc., but we've really only scratched the surface of what the ATARI can do in the area of computation. In this chapter we're going to look particularly at some built-in commands that make certain calculations much, much easier.

One of the first things that we have to understand when we use the ATARI for number crunching is how numbers are expressed on the ATARI. There are two ways:

1. **A whole number in the range −999999999 (9 nines) to 999999999 (again, 9 nines) will be printed as a whole number, just as you would expect.**

2. **A decimal fraction or whole number that is less than −999999999 (9 nines) or greater than 999999999 (9 nines) will be printed in scientific notation as a decimal number between one and ten multiplied by a power of 10. For example, the number 12345678909 would be written 1.23456789E+10, which means: Move the decimal place 10 places to the *right*. You should read the number as "1.23456789 times 10 to the 10th power." The number 1.234E−3 means 0.001234, since the −3 means move the decimal point 3 places to the left.**

Here is a short program that prints out the *factorials* of the integers from 1 to 20. The factorial of a positive integer is the

product of the integer and all of the positive integers less than it. For example, 5 factorial (written 5!) is 5×4×3×2×1, or 120. The point of the program is to show that when the factorials exceed 999999999, they are written in scientific notation.

```
100 REM - Chapter 13, No. 1
110 GRAPHICS 0:PRINT
120 PRINT "N ","N Factorial (N!)"
130 PRINT "--","----------------"
140 FACTORIAL=1
150 FOR N=1 TO 20
160 FACTORIAL=FACTORIAL*N
170 PRINT N,FACTORIAL
180 NEXT N
190 GOTO 190
```

The SQR(X) Function

The SQR(X) function will compute the square root of X, where X represents a non-negative number. X may be a numeric variable [SQR(NUM)], a constant [SQR(25)], or a variable expression [SQR(B*B-4*A*C)]. X is called the *argument* of the function. Here is a program that prints a square and square root table:

```
100 REM - Chapter 13, No. 2
110 GRAPHICS 0:PRINT
120 PRINT "NUMBER","SQUARE","SQUARE ROOT"
130 PRINT "------","------","-----------"
140 FOR K=1 TO 20
150 PRINT K, K*K,SQR(K)
160 NEXT K
170 GOTO 170
```

Note that the variable K is the argument in the SQR(X) function in line 150. See what happens if you change line 140 to

```
140 FOR K=10 TO -10 STEP-1
```

Error message 3, right? The argument in the SQR(X) function cannot be negative.

The ABS(X) Function

The *absolute value* of a number X (written $|X|$) is defined in mathematics as follows:

$$|X| = \begin{cases} X, \text{ if X is positive or zero} \\ -X, \text{ if X is negative} \end{cases}$$

For example, $|5| = 5, |0| = 0$, and $|-5| = 5$. The ATARI function ABS(X) works in exactly the same way. Try this program:

```
100 REM - Chapter 13, No. 3
110 GRAPHICS 0:PRINT
120 PRINT "Type a number"
130 PRINT "(positive, negative, or zero)"
140 INPUT NUM
150 PRINT:PRINT "!";NUM;"! = ";ABS(NUM)
160 PRINT:PRINT
170 GOTO 120
```

The INT(X) Function

The INT(X) function gives the *greatest integer less than or equal to X*. Thus, it gives the whole number part of a decimal fraction. For example, INT(5.23) = 5, INT(0.2345) = 0, and INT(123.56) = 123. However, INT(-1.23) = -2, because -2 is the greatest integer that is *less* than -1.23. (The INT function does not round numbers; 123 is the greatest integer that is less than 123.56.) It might be helpful to remember that INT(X) is *never* to the *right* of X on the number line, as is shown in Figure 13.1.

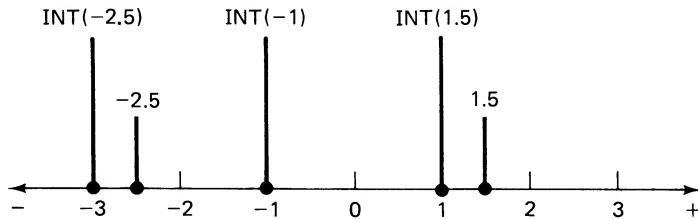

FIGURE 13-1   INT(X) Function

One important use of the INT(X) function is to round numbers. The following program will round a positive number to 2 decimal places.

```
100 REM - Chapter 13, No. 4
110 GRAPHICS 0:PRINT
120 PRINT "Type a number to be rounded."
130 INPUT NUM
140 ROUNDNUM=INT(100*NUM+.5)/100
150 PRINT
160 PRINT NUM;" rounded is ";ROUNDNUM;"."
170 PRINT:PRINT
180 GOTO 120
```

The key to the program is line 140. Before we analyze it, let's recall from arithmetic that multiplying a number by 100 moves the decimal point 2 places *to the right*, and dividing a number by 100 moves the decimal point 2 places *to the left*. Now, suppose the number you picked to round is 5.617. Table 13-1 below shows how ROUNDNUM is calculated in line 140:

TABLE 13-1   Calculation of ROUNDNUM

NUM	100*NUM	100*NUM+.5	INT(100*NUM+.5)	INT(100*NUM+.5)/100
5.617	561.7	562.2	562	5.62

To round a number to one decimal place, replace the 100s in line 140 with 10s. You can easily see how to round off to three, four, or five decimal places. *NOTE:* This won't work for negative numbers; we'll take care of that after we've looked at the SGN(X) function.

A second important use of the INT(X) is to find the remainder when dividing one integer by another. Try this program:

```
100 REM - Chapter 13, No. 5
110 GRAPHICS 0:PRINT
120 PRINT "Type two positive whole numbers."
130 INPUT NUM,DEN
140 QUO=NUM/DEN
150 WHATSLEFT=NUM-INT(QUO)*DEN
160 PRINT
170 PRINT NUM;" divided by ";DEN;" is ";INT(QUO)
180 PRINT "with a remainder of ";WHATSLEFT
190 PRINT:PRINT
200 GOTO 120
```

A variation of this last use of the INT(X) function is included in the exercises at the end of this chapter.

The RND(X) Function

The RND(X) function gives a *random number* that is a 10-place decimal greater than or equal to zero and less than 1. For convenience, we will always use 0 as the argument, but any variable or constant will give similar results. Some possible values for RND(0) are: 0.1234567891, 0.5555555555, 0.9999999999, and 0.1551551551.

The RND(X) function is the heart of all computer "game of chance" programs. Suppose, for example, that we would like to simulate the roll of a die (singular of dice). We need a random

integer between 1 and 6, inclusive. We can generate (create) one with the following BASIC expression:

```
ROLL = INT(6*RND(0))+1
```

Here is the mathematical proof for those who have had some algebra:

IF $0 <= $ RND(0) < 1 , then
$0 = 6 \times 0 <= 6 \times$ RND(0) $< 6 \times 1 = 6$, or
$0 <= 6 \times$ RND(0) < 6 , and
$0 <= $ INT($6 \times$ RND(0)) < 6, that is,
INT($6 \times$RND(0))   is one of the integers (0,1,2,3,4,5), and
INT($6 \times$RND(0))+1 is one of the integers (1,2,3,4,5,6).

Here is a general BASIC formula to pick one of the first N integers at random and store it in the variable NUM. For example, if N is 30, then NUM will be one of the integers (1,2,3,...,28,29,30).

$$NUM = INT(N*RND(0))+1$$

SGN(X) gives the value 1, 0, or –1, depending on whether X is positive, zero, or negative. The possibilities are shown in Table 13-2.

The SGN(X) Function

TABLE 13-2	Possibilities of SGN(X)
X	SGN(X)
positive (X>0)	1
zero (X=0)	0
negative (X<0)	–1

One use of the SGN(X) function is in rounding negative numbers. Recall program Chapter 13, No. 4, and add these lines:

```
100 REM - Addition to Chapter 13, No. 4
135 SIGN=SGN(NUM)
136 NUM=ABS(NUM)
145 ROUNDNUM=SIGN*ROUNDNUM
```

Do you see what these lines do? First, we store the SIGN of the number to be rounded as 1, 0 or –1 (line 135). Then we take the absolute value of the number (line 136) so that we are work-

ing with a non-negative number. Then, after we've rounded this non-negative number, we multiply it by SIGN to give it its original algebraic sign. Notice that line 145 will change the value of ROUNDNUM only if our original number was negative.

The LOG(X) Function (Advanced Math)

The LOG(X) function gives the *natural logarithm* of the number X. The *base* of natural logarithms is the number e, which to 10 decimal places is 2.7182818285. If Y=LOG(X), then e raised to the Yth power will equal X, or $e^Y=X$.

The CLOG(X) Function (Advanced Math)

The CLOG(X) function gives the *common logarithm* of the number X. The base of common logarithms is 10. If Y=CLOG(X), then 10 raised to the Yth power will equal X, or $10^Y=X$.

The EXP(X) Function (Advanced Math)

The EXP(X) function gives the number e raised to the Xth power, or $EXP(X) = e^X$. Note that

$$if\ Y\ =\ LOG(X)\ ,\ then\ X\ =\ EXP(Y)$$

The Trigonometric Functions (Advanced Math)

ATARI BASIC provides the following three trigonometric functions:

SIN(X)   (sine of X)
COS(X)   (cosine of X)
ATN(X)   (inverse tangent [arctangent] of X)

The argument X [for SIN(X) and COS(X)] is normally in radians, but we can change it to degrees by giving the command DEG somewhere near the beginning of the program before we use the trigonometric functions. If we should later want to change to radians, we would give the command RAD. The argument X for the function ATN(X) is any real number, and the value of the function is either in degrees or radians, depending on whether we have given the DEG command or not. Values of other trigonometric functions that are not included in ATARI BASIC can be derived as follows:

Tangent of X: TANX=   SIN(X)/COS(X)
                   [COS(X) not equal to 0]
Secant of X: SECX = 1/COS(X)
                   [COS(X) not equal to 0]

Cosecant of X: CSCX = 1/SIN(X)
                   [SIN(X) not equal to 0]

Because the tangent of pi/4 radians (45 degrees) is equal to 1, the following command will calculate a value for pi that is accurate to six decimal places. Try it!

```
PI=4*ATN(1)
```

We close this chapter with a program that uses the SIN(X) and COS(X) functions to draw a circle. If you have studied trigonometry, you will recall that if we have a circle with center O (**0,0**) and radius 1, then a point P on the circle such that the angle between the radius vector OP and the positive X-axis has a measure of A degrees (measured counter-clockwise from the positive X-axis) will have coordinates [cos(A),sin(A)]. (Well, perhaps you didn't recall that fact. Anyway, the program works!)

```
100 REM - Chapter 13, No. 6
110 GRAPHICS 7+16
120 COLOR 1
130 DEG
140 FOR A=0 TO 360 STEP 3
150 X=COS(A):X=INT(46*X+.5)
160 Y=COS(A):Y=INT(40*Y+.5)
170 PLOT 80+X,48-Y
180 NEXT A
190 GOTO 190
```

Note that line 140 gives the following values for the angle A: 0,3,6,9,...,351,354,357,360. At lines 150 and 160, the X- and Y-coordinates for a point on the circle are

1. **Calculated using the COS(X) and SIN(X) functions.**

2. **Multiplied by a scale factor (46 for X, 40 for Y, because the pixels are slightly taller than they are wide) to make the circle fill the screen.**

3. **Rounded to the nearest integer.**

At line 170, the point is plotted, using the point **80,48** as the center of the circle. (Remember that GRAPHICS 7+16 has 160 columns and 96 rows, so the point **80,48** is in the center of the screen.)

EXERCISES

1. Evaluate each of the following expressions:

   INT(4)   _____

   INT(23.75)   _____

   INT(-13/2)   _____

   INT(13)/2   _____

   INT(-.999999)   _____

   INT(-3.56)   _____

   INT(0.378)   _____

   INT(-5.01)   _____

2. Write the printout of the following program:

```
100 FOR K=2 TO 10
110 IF 20/K=INT(20/K) THEN ? K;" is a factor of 20."
120 NEXT K
```

3. List the set of possible values for R in each of the following statements:

   R = INT(3*RND(0))   _____

   R = INT(3*RND(0))+1   _____

   R = INT(10*RND(0))   _____

   R = 3*INT(RND(0))   _____

   R = INT(5*RND(0))-5   _____

   R = 2*INT(2*RND(0))-1   _____

   R = 10-INT(5*RND(0))   _____

   R = 1/(INT(3*RND(0))+1)   _____

4. Write a BASIC statement that would generate each of the following sets of numbers (not necessarily in the same order)

(1,2,3,4,5,6)   _____

(-2,-1,0,1,2)   _____

(5,10,15,20)   _____

(.5,1,1.5,2,2.5)   _____

(5,8,11,14,17,...)   _____

(-50,0,50)   _____

5. Write an original program that uses the Pythagorean Theorem (in a right triangle, the sum of the squares of the two legs is equal to the square of the hypotenuse) to calculate the hypotenuse if the user types in the lengths of the two legs. Here is a sample run:

```
RUN

Type in the two legs of a right
triangle; for example,  3,4
?5,12

The hypotenuse is 13.

READY
■
```

6. An integer greater than 1 is *prime* if it has no factors other than itself and 1. Write an original program that will determine whether or not a given number is prime.

14

String Functions

Just as we learned about some built-in numeric functions in the last chapter, in this chapter we're going to look at some built-in string functions, which greatly simplify our work with strings.

Introduction

The LEN(A$) function calculates the length (number of characters) of the string A$. For example,

The LEN(A$)
Function

if A$ = "TESTING", then LEN(A$) = 7
if A$ = "ATARI COMPUTER", then LEN(A$) = 14
if A$ = "The price is $97.50.", then LEN(A$) = 20
if A$ = "   2", then LEN(A$) = 3

Try this simple program to see how LEN(A$) works:

```
100 REM - Chapter 14, No. 1
110 DIM A$(40)
120 GRAPHICS 0:PRINT
130 PRINT "Type a word or message"
140 PRINT "not longer than one line."
150 INPUT A$
160 L=LEN(A$)
170 PRINT:PRINT A$
180 PRINT "has ";L;" characters."
190 PRINT:PRINT:GOTO 130
```

Note that the argument for the LEN(A$) function may also be a *string literal*. For example,

LEN("TESTING") = 7,
LEN("ATARI COMPUTER") = 14,
LEN("$125.95") = 7, and
LEN("This is a string literal.") = 25.

The LEN(A$) function is often used with FOR...NEXT loops to check for a certain character in a string. Here is a program that will print the user's name in the order *last name, first name* if he types it in the opposite order.

```
100 REM - Chapter 14, No. 2
110 DIM NAME$(30)
120 GRAPHICS 0:PRINT
130 PRINT "Type your name:  FIRST LAST"
140 PRINT "Example:  Sammy Smith"
150 INPUT NAME$
160 L=LEN(NAME$)
170 FOR K=1 TO L
180 IF NAME$(K,K)=" " THEN 210
190 NEXT K
200 PRINT "First and last name, please.":GOTO 130
210 PRINT:PRINT NAME$(K+1,L);", ";NAME$(1,K-1)
220 PRINT:PRINT:GOTO 130
```

Do you see how it works? The FOR...NEXT loop finds the space between the first and last names. This is the Kth character. Then the last name must contain the characters K+1 (the next one after the space) to L (the last character, or the length of the whole name). The first name contains the characters 1 to K-1 (the character before the space). At line 210 we print the last name, a comma and a space, and then the first name. Line 200 gives the user another chance in case he or she didn't leave a space after his or her first name.

Here's another example of using the LEN(A$) function:

```
100 REM - Chapter 14, No. 3
110 DIM NAME$(30)
120 GRAPHICS 0:PRINT
130 PRINT "Type your name."
140 INPUT NAME$
150 L=LEN(NAME$)
160 FOR K=L TO 1 STEP -1
170 PRINT NAME$(K,K);
180 NEXT K
190 PRINT:PRINT
200 GOTO 130
```

See what happens if you remove the semicolon at the end of line 170.

We can use a similar loop technique to fill a string with a
particular character. Suppose we would like to print a row of
exactly 35 asterisks (*) across the screen. We could use the fol-
lowing program:

```
100 REM - Chapter 14, No. 4
110 DIM A$(35)
120 FOR K=1 TO 35
130 A$(K,K)="*"
140 NEXT K
150 PRINT A$
```

Alternatively, we could do this:

```
100 REM - Chapter 14, No. 5
110 FOR K=1 TO 35
120 PRINT "*";
130 NEXT K
140 PRINT
```

The advantage of the first method is that the row of asterisks
is stored in a string and we can print it as many times as we
want, without going through the loop again. Moreover, we can
print a row of *any number* (less than or equal to 35) of asterisks.
For example, if we wanted a row of 10 asterisks, a command
might be

```
210 PRINT A$(1,10)
```

Now suppose we combine the idea of a string of asterisks
with the LEN(A$) function to dress up a title for a program. Try
this program:

```
100 REM - Chapter 14, No. 6
110 DIM A$(38),TITLE$(38)
120 FOR K=1 TO 38:A$(K,K)="*":NEXT K
130 GRAPHICS 0:PRINT
140 PRINT "What is the title"
150 INPUT TITLE$
160 L=LEN(TITLE$)
170 GRAPHICS 0:PRINT
180 PRINT A$(1,L)
190 PRINT TITLE$
200 PRINT A$(1,L)
```

It is often useful to fill a string with spaces; then we can use it as a sort of tabulator. Suppose we fill a string called BLANK$ with exactly 38 spaces (the width of the screen). Then if we would like to print the string NAME$ indented 5 spaces from the left margin, we could write

```
190 PRINT BLANK$(1,5);NAME$
```

This would be a good place to spend some time making your own experiments with the LEN(A$) function and with filling strings with certain characters using FOR...NEXT loops.

The STR$(X) Function

The STR$(X) function converts the contents of the numeric variable X to a string so that the digits of the number can be treated as characters, not digits. The following program gives an example.

```
100 REM - Chapter 14, No. 7
110 DIM NUM$(10)
120 GRAPHICS 0:PRINT
130 PRINT "Type a 3-digit number."
140 INPUT NUM
150 NUM$=STR$(NUM)
160 L=LEN(NUM$)
170 IF L<>3 THEN 130
180 PRINT "The hundreds digit is ";NUM$(1,1)
190 PRINT "The tens    digit is ";NUM$(2,2)
200 PRINT "The units   digit is ";NUM$(3,3)
```

If we convert a numeric variable to a string variable, then we can use the LEN(A$) function to see how many digits the number has. This gives us a way to print numbers in a column so that the decimal points line up. Here is a modification of our factorial program from Chapter 13 to illustrate this idea:

```
100 REM - Chapter 14, No. 8
110 DIM NUM$(10),BLANK$(10)
120 FOR K=1 TO 10:BLANK$(K,K)=" ":NEXT K
130 GRAPHICS 0:PRINT
140 PRINT "N ","N Factorial"
150 PRINT "--","-----------"
160 FACTORIAL=1
170 FOR N=1 TO 13
180 FACTORIAL=FACTORIAL*N
```

```
190 NUM$=STR$(FACTORIAL):L=LEN(NUM$)
200 PRINT N,BLANK$(1,11-L);FACTORIAL
210 NEXT N
220 GOTO 220
```

The VAL(A$) function works just the opposite of the STR$(X) function: It gives the numerical value of the number that is stored as A$. Consider these examples:

The VAL(A$) Function

if A$ = "2.19",   then VAL(A$) = 2.19.
if A$ = "714Q", then VAL(A$) = 714.
if A$ = "3PO",   then VAL(A$) = 3, but
if A$ = "R2D2", then ERROR MESSAGE 18 results.

Think of it as follows: The VAL(A$) function looks at the characters in the string A$ column by column from left to right. If the first character is not a digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), decimal point (.), or sign (+, -), then error message 18 results, as in the last example. Otherwise, the longest sequence of meaningful numeric characters becomes the value of VAL(A$). Here's a simple test program to see how VAL(A$) works:

```
100 REM - Chapter 14, No. 9
110 DIM A$(15)
120 GRAPHICS 0:PRINT
130 PRINT "Type a numeric expression."
140 INPUT A$
150 PRINT
160 PRINT "VAL('";A$;"') = ";VAL(A$)
170 PRINT:PRINT:GOTO 130
```

Every character that can be printed on the screen is represented internally in the computer by a number. The list of characters and their associated numbers is a CODE; A=64, B=65, C=66, etc. This code is known as ASCII, for the American Standard Code for Information Interchange. The ATARI version of this code is called ATASCII (ATARI ASCII). Table 14-1 gives the ATASCII decimal code for the characters that will print on most printers. Using the ATASCII codes is the topic of the next two sections.

The ASCII and ATASCII Codes

TABLE 14-1   ATASCII Character Set

ATASCII Decimal Code	Character	ATASCII Decimal Code	Character	ATASCII Decimal Code	Character
32	(space)	64	@	96	♦
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	♠
60	<	92	\	124	|
61	=	93	]	125	(clear screen)
62	>	94	∧		
63	?	95	_		

Notice that the codes for the lowercase letters are exactly 32 more than the codes for the uppercase letters. For example, the ATASCII code for A is 65, and the code for a is 65+32, or 97. Likewise, the code for Z is 90, and the code for z is 90+32, or 122.

The codes for the control graphics characters (which normally don't print on the printer) are numbers 1 through 26. The codes for the graphics symbols for the playing card suits are as follows:

♠	spade	123
♥	heart	0
♣	club	16
♦	diamond	96

The ATASCII code for the inverse video representation of a character is always 128 more than the code for the character. For example, the code for A is 90, and the code for "inverse video A" is 90+128, or 218.

The CHR$(X) function gives the character whose ATASCII code is the number X. For example the command

```
210 PRINT CHR$(65);CHR$(66);CHR$(67)
```

will print ABC. Here is a simple program you can use to look at the characters that correspond to given ATASCII numbers:

```
100 REM - Chapter 14, No. 10
110 PRINT CHR$(125)
120 PRINT "Starting ATASCII Code Number ";:INPUT START
130 PRINT "Ending ATASCII Code Number   ";:INPUT FINISH
140 FOR K=START TO FINISH
150 PRINT K,CHR$(K)
160 NEXT K
```

Note the command at line 110 to clear the screen. Use the program to look at the control graphics characters (input 1 and 26), the inverse video uppercase letters (input 193 and 218), and the inverse video lowercase letters (input 225 and 250).

The following program picks a random number between 65 and 90 (line 160) to choose one of the letters of the alphabet, and also picks a random place to print it on the GRAPHICS 2+16 screen.

```
100 REM - Chapter 14, No. 11
110 GRAPHICS 2+16
120 X=INT(18*RND(0))+1
130 Y=INT(10*RND(0))+1
140 C=INT(4*RND(0))
150 IF C>1 THEN C=C+2
160 R=INT(26*RND(0))+65+32*C
170 POSITION X,Y
180 PRINT #6;CHR$(R)
190 GOTO 120
```

But how did we get the different colors? Refer for a moment to Table 3-6 "Character Types and SETCOLOR Numbers." The color of a character in GRAPHICS 1 or 2 depends on whether the character is upper- or lowercase, normal or inverse video. From the ATASCII Character Set chart we know that lowercase ATASCII numbers are 32 more than uppercase; uppercase

inverse video 128 more than uppercase; and lowercase inverse video 32+128=160 more than uppercase. So, if we add either 0, 32, 128, or 160 to the character code, we will get orange, light green, blue, or red letters. We note that the numbers 0, 32, 128, and 160 are all multiples of 32: 0=0×32, 32=1×32, 128=4*32, and 160=5*32. We need to pick one of the numbers 0, 1, 4, or 5 at random and add 32 times that number to the character code at line 160. This is the function of lines 140 and 150. Now, if you promise not to drive your teacher crazy, add some music to the program:

```
185 S=C:IF S>1 THEN S=S-2
186 SOUND S,25*Y,10,6
```

The ASC(A$) Function

The ASC(A$) function gives the ATASCII code for the character stored in the string A$; it is just the reverse of the CHR$(X) function. Try this program:

```
100 REM - Chapter 14, No. 12
110 DIM NAME$(30),A$(1)
120 GRAPHICS 0:PRINT
130 PRINT "Type your name."
140 INPUT NAME$:L=LEN(NAME$)
150 PRINT
160 FOR C=1 TO L
170 A$=NAME$(C,C)
180 PRINT A$;" - ";ASC(A$)
190 NEXT C
```

The next program uses both the CHR$(X) and ASC(A$) functions to change uppercase letters in a name to lowercase.

```
100 REM - Chapter 14, No. 13
110 DIM NAME$(30)
120 GRAPHICS 0:PRINT
130 PRINT "Type your first and last names."
140 INPUT NAME$:L=LEN(NAME$)
150 FOR C=1 TO L
160 IF NAME$(C,C)=" " THEN 190
170 NEXT C
180 GRAPHICS 0:PRINT:GOTO 130
190 PRINT
200 PRINT NAME$(1,1);
210 FOR K=2 TO C-1
```

```
220 PRINT CHR$(ASC(NAME$(K,K))+32);
230 NEXT K
240 PRINT " ";NAME$(C+1,C+1);
250 FOR K=C+2 TO L
260 PRINT CHR$(ASC(NAME$(K,K))+32);
270 NEXT K
280 PRINT:PRINT:PRINT:GOTO 130
```

We already know that we can refer to part of a string:

if A$** = **"TESTING", then A$(4,6) = **"TIN"**

See if you can find out what A$(3) means. Right! A$="STING", the part of A$ from the third character through the last character. Thus, A$(N) is an easier way to write A$(N,LEN(A$)). We can use this idea to *concatenate*, or "add," strings together. Try this program:

```
100 REM - Chapter 14, No. 14
110 DIM A$(100),B$(20),C$(20)
120 A$="1st STRING."
130 B$="2nd STRING."
140 C$="3rd STRING."
150 A$(LEN(A$)+1)=B$
160 A$(LEN(A$)+1)=C$
170 PRINT A$
```

Here is a program that gives a faster way to fill a string with a particular character (see program Chapter 14, No. 3)—"*," for example. (Please don't ask us why it works!)

```
100 REM - Chapter 14, No. 15
110 DIM A$(30)
120 A$(1)="*"
130 A$(30)="*"
140 A$(2)=A$:PRINT A$
```

Try this program for a little fun:

```
100 REM - Chapter 14, No. 16
110 DIM A$(37)
120 FOR K=1 TO 37:R=INT(26*RND(0)):A$(K,K)=CHR$(R):NEXT K
125 PRINT A$
130 FOR N=2 TO 37:PRINT A$(N);A$(1,N-1):NEXT N:GOTO 125
```

EXERCISES

1. Complete each of the following expressions:

    If Y$ = "LENGTH FUNCTION", then LEN(Y$) = _____

    If Q$ = "QUOTIENT OF A and B", then LEN(Q$) = _____

    If P$ = "2 × 3 = 6", then LEN(P$) = _____

    If S$ = "   SPACE   ", then LEN(S$) = _____

    If B$ = "BASIC for BEGINNERS", then LEN(B$) = _____

    If X$ = "$623.98", then LEN(X$) = _____

2. Write the printout of the following program:

```
100 DIM A$(30)
110 A$="ALFRED NOYES WROTE POETRY."
120 PRINT A$(10,12)
130 PRINT A$(8,9)
140 L=LEN(A$)
150 PRINT A$(L-3,L-1);" ";A$(1,1);" ";A$(4,7);A$(20,23)
```

3. Complete the following table:

A$	LEN(A$) (L)	A$(3,3)	A$(2,4)	A$(L-3,L)
AMES HIGH SCHOOL	_____	_____	_____	_____
CALL ME MADAM	_____	_____	_____	_____
_____	___4___	_____	_____	__LIST__
4 JULY 1776	_____	_____	_____	_____
81 JONES SAM	_____	_____	_____	_____
_____	_____	_____	__ASI__	__ASIC__

4. Write an original program that will produce a printout like the sample RUN.

```
RUN

Type in your full name.
?SPANGLER ARLINGTON BROUGH
```

```
SPANGLER, your initials are S A B.

READY
■
```

5. Write an original program that will produce a printout like the sample "RUN."

```
RUN

Type your first name.
?HESPATIA

HESPATIA
ESPATIAH
SPATIAHE
PATIAHES
ATIAHESP
TIAHESPA
IAHESPAT
AHESPATI
HESPATIA

READY
■
```

6. Write an original program that will produce a printout like the sample "RUN."

```
RUN

Type your full name.
?JOHN HUGHES CALVIN

Without vowels, your name is
JHN HGHS CLVN

READY
■
```

7. Suppose that A$ = "TEST23A", B$ = "7.SEPTEMBER.1982", and ABC = 55. Evaluate each of the following expressions. If the expression produces an error, write "ERROR." Try to do this exercise *without* the computer.

    CHR$(VAL(CHR$(48)))   _____

    STR$(ABC)   _____

    VAL(CHR$(VAL("48")))   _____

    STR$("ABC")   _____

VAL(2.95)   _____

CHR$(VAL(B$(14,15)))   _____

VAL(A$(5))   _____

CHR$(ABC)   _____

VAL(STR$(ABC))+VAL(B$(13,14))   _____

ASC(STR$(VAL(C$(1))))   _____

ASC(A$(1))   _____

ASC(VAL(A$(5)))   _____

ASC(CHR$(VAL(A$(5))+41))   _____

VAL("LOVE")   _____

ASC("QUESTION")   _____

STR$ (A$)   _____

CHR$(A$(5,6))   _____

STR$(149XYZ)   _____

STR$(XYZ149)   _____

CHR$((VAL(C$(14,15))+VAL(C$(15,16)))/2)   _____

8. Write an original program that will produce a printout like the sample "RUN."

```
RUN

Type in your first name.
?ALPHONSE

You are spaced out, A-L-P-H-O-N-S-E.

READY
■
```

9. Write an original program that will center a title on the screen, as in this sample
    "RUN."

```
---------------------------------------------
|                                           |
|   RUN                                     |
|                                           |
|   Type in the title to be centered.       |
|   ?BASIC for BEGINNERS                     |
|                                           |
|            BASIC for BEGINNERS             |
|                                           |
```

10. Write an original program that will reprint a short message that the user types in, but in *lowercase inverse video*.

15

Variables With One Subscript (Arrays)

Suppose that we would like to keep track of the points-per-game of 20 basketball players. We would need to use 20 different variable names, 20 different computation statements, etc., etc. Fortunately, there is an easier way—using subscripted variables.

If we pursue the problem of the basketball players, we might think of using 20 variable names such as PPG1, PPG2, PPG3, etc. The variable names would be similar to each other, but still we would have 20 distinct variable names. We can, however, use these names: PPG(1), PPG(2), PPG(3), ..., PPG(19), PPG(20). In this form, the variables are *subscripted variables*, and the numbers 1, 2, 3,...,19, 20 in parentheses are the *subscripts*. To use subscripted variables, we must first indicate in a DIMension statement *how many* variables we want to use. For the basketball players, we could write:

```
110 DIM PPG(20)
```

Note that this is similar to DIMensioning a string variable. Now suppose we read 20 pieces of data for the players and store them in the variables PPG(1) to PPG(20). Here's the program:

```
100 REM - Chapter 15, No.1
110 DIM PPG(20)
120 FOR N=1 TO 20
130 READ POINTS
140 PPG(N)=POINTS
150 NEXT N
```

```
160 GRAPHICS 0
170 PRINT "PLAYER","POINTS per GAME"
180 PRINT "------","----------------"
190 FOR N=1 TO 20
200 PRINT "   ";N,"           ";PPG(N)
210 NEXT N
220 GOTO 220
300 DATA 7,11,9,15,4,22,6,13,10,8
310 DATA 13,2,5,9,11,4,8,19,6,7
```

Notice what happens: As we READ the data in the FOR... NEXT loop in lines 120 to 150, we let the variable N be the subscript for the variable PPG. When N is 1, then PPG(1) gets the value 7. When N is 2, then PPG(2) gets the value 11. When N is 11, then PPG(11) gets the value 13 (the first data value in line 310). Run the program again and compare the chart that is printed with the data in lines 300 and 310. We have assigned a number from 1 to 20 to each of the data items. This number is the subscript for the variable PPG.

Look at lines 130 and 140. First we READ a data value into a variable called POINTS (line 130), and then we transfer this value into the variable PPG(N) (line 140). In ATARI BASIC, we cannot "read" values directly into subscripted variables. We could, however, combine the two lines into one, as below:

```
130 READ POINTS:PPG(N)=POINTS
```

Arrays

An *array* is simply the list of numbers assigned to a subscripted variable. Our array from the previous program looks like this (Figure 15-1).

PPG(0)	?
PPG(1)	7
PPG(2)	11
PPG(3)	9
PPG(4)	15
PPG(18)	19
PPG(19)	6
PPG(20)	7

FIGURE 15-1   Points Per Game Array

Notice the freebie in our array: There is an element of the array labeled PPG(0). We indicated its value with a ?, because our program didn't store anything there. You can find out what PPG(0) contains by typing PRINT PPG(0) and pressing the RETURN key.

We remember from Chapter 4 that numeric variables are always initialized to zero. For example, if we turn the computer on with the BASIC cartridge inserted and type

```
PRINT A,B,SUM
```

and then press RETURN, the result on the screen will be

```
O           O           O

READY
■
```

However, if we type

```
DIM A(3):PRINT A(1):PRINT A(2):PRINT A(3)
```

and then press RETURN, the result might look something like this:

```
OE-;O
2.0436817E-37
-6.>0004E-<5

READY
■
```

Thus, subscripted variables are NOT automatically initialized to zero; they may contain GARBAGE. Therefore, they should always be initialized to some value, usually zero. Here is a program to set each of 10 subscripted variables named ARRAY equal to zero:

```
110 DIM ARRAY(10)
120 FOR N=1 TO 10
130 ARRAY(N)=0
140 NEXT N
```

This could be condensed to

Initializing an Array

```
110 DIM ARRAY(10)
120 FOR N=1 TO 10:ARRAY(N)=0:NEXT N
```

The following program "shuffles" the integers from 1 to 20 and prints them in shuffled order. The idea is to randomly pick a number from 1 to 20 with the RND(X) function, and to do this 20 times. However, we need a way to prevent picking the same number twice. We use the array N to keep track of which numbers have been picked. We start by setting all of the elements of N equal to zero. Then, if we pick the random number 13, we set N(13) equal to 1, to show that 13 has been picked. If we pick 13 again, then we have to go back and try again.

```
100 REM - Chapter 15, No. 2
110 DIM N(20)
120 FOR K=1 TO 20
130 N(K)=0
140 NEXT K
150 GRAPHICS 0:PRINT
160 FOR K=1 TO 20
170 NUM=INT(20*RND(0))+1:REM--PICK A RANDOM NUMBER 1-20
180 IF N(NUM)=1 THEN 170:REM--WE'VE ALREADY PICKED NUM
190 PRINT K,NUM
200 N(NUM)=1:REM--SHOW THAT WE PICKED NUM
210 NEXT K
```

You can easily modify this program to shuffle the order of any list.

Storing Computations in Arrays

In Chapter 13 we discussed a program (Chapter 13, No. 6) to draw a circle. You may remember that the program was relatively slow because a certain amount of calculation was involved. The next program speeds up the drawing by calculating the values and *storing them in an array*, then reading them out of the array to draw several circles. If you enjoy working with graphics, you can think of many modifications of this program: pick a random point for the center, pick a random radius, etc.

```
100 REM - Chapter 15, No. 3
110 DIM CIRX(360),CIRY(360):DEG
120 GRAPHICS 0:PRINT "Graphing begins at count 360."
130 FOR A=0 TO 360
140 X=COS(A):CIRX(A)=X
150 Y=SIN(A):CIRY(A)=Y
160 PRINT A;" ";
170 NEXT A
180 GRAPHICS 7:COLOR 1
190 FOR CENX=25 TO 135 STEP 5
200 FOR A=0 TO 360 STEP 5
210 X=CIRX(A):X=INT(23*X+.5)
220 Y=CIRY(A):Y=INT(20*Y+.5)
230 PLOT CENX+X,40-Y
```

```
240 NEXT A
250 NEXT CENX
```

Note that at lines 210 and 220 we read the values back out of the arrays CIRX and CIRY, then multiply them by values for the radius and round them to the nearest integer. You can change the circles to ovals by changing the 20 in line 220 to 35. You can also change the step at line 190. We're plotting points every 5 degrees around the circle; it will go faster (fewer points to plot) with a step of 10, and slower (more points to plot) with a step of 3. Did you note the nested FOR...NEXT loops? The outer loop moves the center from left to right, and the inner loop draws the circle.

Here is another example of storing values in arrays; this is a musical one:

```
100 REM - Chapter 15, No. 4
110 DIM PITCH(8)
120 FOR K=1 TO 8
130 READ TONE:PITCH(K)=TONE
140 NEXT K
150 PRINT "Type a number from 1 to 8 to"
160 PRINT "play a tone.  1=do, 2=re, 3=mi, etc."
170 INPUT NUM
180 SOUND 0,PITCH(NUM),10,10
190 PRINT:GOTO 150
200 DATA 121,108,96,91,81,72,64,60
```

The following program reads five grades and finds their average, then shows how much above (+) or below (-) the average each grade is. Storing the grades in an array simplifies the problem.

Applications of Subscripted Variables

```
100 REM - Chapter 15, No. 5
110 DIM GR(5)
120 FOR K=1 TO 5
130 READ GRADE:GR(K)=GRADE:TOTAL=TOTAL+GRADE
140 NEXT K
150 AVE=TOTAL/5
160 GRAPHICS 0:PRINT
170 PRINT "Average of 5 grades is ";AVE
180 PRINT
190 FOR K=1 TO 5
200 PRINT GR(K),GR(K)-AVE
210 NEXT K
220 DATA 85,92,73,65,98
```

Note that we total the grades as we read them in at line 130.

We can use subscripted variables to total more than one item. Suppose we would like to tally the votes for four candidates in the election for student body president. This time we'll use the INPUT statement, and type 1, 2, 3, 4, depending on which of the four candidates is being chosen. We use an input of -1 to stop.

```
100 REM - Chapter 15, No. 6
110 DIM CAND(4)
120 FOR K=1 TO 4:CAND(K)=0:NEXT K
130 GRAPHICS O:PRINT
140 PRINT "INSTRUCTIONS:   At each number, type"
150 PRINT "the number of the candidate (1-4) and"
160 PRINT "press the RETURN key."
170 PRINT "Type -1 to stop."
180 COUNT=1:PRINT
190 PRINT COUNT;" ";:INPUT VOTE
200 IF VOTE=-1 THEN 260
210 IF VOTE<1 OR VOTE>4 THEN 190
220 CAND(VOTE)=CAND(VOTE)+1
230 COUNT=COUNT+1
240 GOTO 190
250 REM *** PRINT RESULTS ***
260 GRAPHICS O:PRINT
270 PRINT COUNT-1;" votes were tallied."
280 PRINT
290 FOR K=1 TO 4
300 PRINT "Candidate ";K;" - ";CAND(K)
310 NEXT K
```

This program can easily be modified to tally the votes for any number of candidates.

Sorting

One of the classic computer problems is to sort a list of numbers into numerical order, or to sort a list of names into alphabetical order. There are many techniques for computer sorting; some of them are quite complicated. We'd like to discuss two that are easy to understand. First, we should understand the term *algorithm*. An algorithm is a set of steps or a procedure to perform a task—usually a repetitive task. A good example of an arithmetic algorithm is long division. Remember? ...trial divisor, write digit for quotient, multiply, subtract, bring down next digit, trial divisor, etc. Every sort program is based on some algorithm, or procedure, for sorting.

Before we look at the first sorting program, let's walk through the procedure. It would be helpful here if you made six small cards or pieces of paper with the numbers 1 through 6 written on them. Arrange the cards in this order:

<p style="text-align:center">4   1   2   3   6   5</p>

Here is our algorithm:

1. **Compare the cards in consecutive pairs (1-2, 2-3, 3-4, 4-5, 5-6).**

2. **If the second number in a pair is *larger* than the first number in the pair, then *exchange* the numbers (cards) in the pair,**

and *start over*. If the second number in the pair is *smaller* than the first number, then continue comparing.

3. If no exchanges are made in a pass, then the numbers (cards) are in order and the procedure is finished.

Now try it with your cards. Here are the results after each operation (* indicates numbers exchanged):

TABLE 15-1		Bubble Sort Diagram				
4	1	2	3	6	5	(original list)
4	2*	1*	3	6	5	(1st pass)
4	2	3*	1*	6	5	(2nd pass)
4	3*	2*	1	6	5	(3rd pass)
4	3	2	6*	1*	5	(4th pass)
4	3	6*	2*	1	5	(5th pass)
4	6*	3*	2	1	5	(6th pass)
6*	4*	3	2	1	5	(7th pass)
6	4	3	2	5*	1*	(8th pass)
6	4	3	5*	2*	1	(9th pass)
6	4	5*	3*	2	1	(10th pass)
6	5*	4*	3	2	1	(11th pass)

Finally! The list is sorted! Now, naturally we don't want to sort many lists by shuffling cards, but remember that the computer can do these things quickly. If we understand the algorithm for sorting, then we can write a program to do it. We know that we can test if one number is larger than another, but how do we *exchange* the values of two variables? Try the following program:

```
100 REM - Chapter 15, No.7
110 A=5:B=10
120 PRINT "A = ";A
130 PRINT "B = ";B
140 PRINT:PRINT "exchange A and B"
150 PRINT
160 TEMPORARY=A
170 A=B
180 B=TEMPORARY
190 PRINT "A = ";A
200 PRINT "B = ";B
```

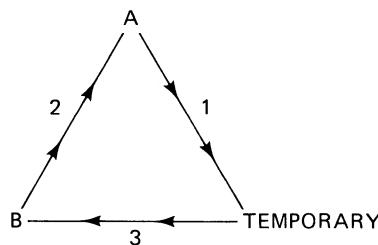The following diagram may help explain the exchanging procedure:

FIGURE 15-2   Exchanging Procedures

Read the diagram clockwise:

1. **Store the contents of A in TEMPORARY (line 160)**
2. **Store the contents of B in A (line 170)**
3. **Store the contents of TEMPORARY in B (line 180)**

Now let's put these ideas together in a program to sort the six numbers we started with on the last page:

```
100 REM - Chapter 15, No. 8 - "BUBBLE" SORT
110 DIM N(6)
120 REM *** STORE LIST TO BE SORTED IN AN ARRAY ***
130 FOR K=1 TO 6
140 READ NUM:N(K)=NUM
150 NEXT K
160 GRAPHICS 0:PRINT
170 REM *** BEGIN SORT ***
180 FOR K=1 TO 5
190 IF N(K+1)>N(K) THEN TEMP=N(K):N(K)=N(K+1):
N(K+1)=TEMP:GOTO 180
200 NEXT K
210 REM *** PRINT SORTED LIST ***
220 FOR K=1 TO 6
230 PRINT K, N(K)
240 NEXT K
300 DATA 4,1,2,3,6,5
```

The sorted list is printed in descending order; if we want it in ascending order, we can change line 220 to

```
220 FOR K=6 TO 1 STEP-1
```

Can you make the appropriate changes in the program so that you can sort a list of 10 or 20 or even 50 numbers?

The sorting algorithm in the previous program is called a *bubble sort* because the larger numbers "bubble" to the front of the list on successive passes. You can see this bubbling process for the numbers 5 and 6 in the chart of the exchanges Table 15-1; note how 6 and then 5 bubble to the front of the list.

The second type of sort that we'll consider is called a *replace-ment sort*. The idea is: Search the list to find the largest number, then search the rest of the list to find the next largest number, then search the rest of the list to find the third largest number, etc. We can write the algorithm as follows.

1. **Compare the first number in the list pairwise with each of the following numbers. If any of the following numbers is *larger* than the first one, then *exchange* these two. At the end of this pass, the largest number will be first in the list.**

2. **Repeat this process, comparing the second number in the list with each of the following numbers. At the end of this pass, the second largest number will be second in the list.**

3. **Repeat this process until you have compared all but the last number in the list to each of the following numbers.**

Here are the results (Table 15-2) of each pass if you try it with the cards you made. We strongly recommend actually moving the cards; this is the best way to see how a sort algorithm works.

TABLE 15-2		**Replacement Sort**				
		Diagram				
4	1	2	3	6	5	(original list)
6*	1	2	3	4*	5	(1st pass)
6	2*	1*	3	4	5	(2nd pass)
6	3*	1	2*	4	5	(3rd pass)
6	4*	1	2	3*	5	(4th pass)
6	5*	1	2	3	4	(5th pass)
6	5	2*	1	3	4	(6th pass)
6	5	3*	1	2	4	(7th pass)
6	5	4*	1	2	3*	(8th pass)
6	5	4	2*	1*	3	(9th pass)
6	5	4	3*	1	2*	(10th pass)
6	5	4	3	2*	1*	(11th pass)

The following program for the replacement sort has much in common with the previous program for the bubble sort, but note that we need two nested FOR...NEXT loops instead of a single one. The J, or outer, loop indicates the first number in the pair that we are comparing, and the K, or inner, loop indicates the second number in the pair. The exchange procedure is the same as before.

```
100 REM - Chapter 15, No. 9 - REPLACEMENT SORT
110 DIM N(6)
```

```
120 REM *** STORE LIST TO BE SORTED IN AN ARRAY ***
130 FOR K=1 TO 6
140 READ NUM:N(K)=NUM
150 NEXT K
160 GRAPHICS 0:PRINT
170 REM *** BEGIN SORT ***
180 FOR J=1 TO 5
190 FOR K=J+1 TO 6
200 IF N(K)>N(J) THEN TEMP=N(J):N(J)=N(K):N(K)=TEMP
210 NEXT K
220 NEXT J
230 REM *** PRINT SORTED LIST ***
240 FOR K=1 TO 6
250 PRINT K, N(K)
260 NEXT K
300 DATA 4,1,2,3,6,5
```

Often it is more convenient to have the user input the numbers to be sorted, rather than place them in a DATA statement. The following modification of the replacement sort program lets you input as many numbers as you like to be sorted.

```
100 REM - Chapter 15, No. 10 - REPLACEMENT SORT (UNIVERSAL)
110 GRAPHICS 0:PRINT
120 PRINT "How many items to be sorted ";:INPUT MAX
130 DIM N(MAX)
140 REM *** STORE LIST TO BE SORTED IN AN ARRAY ***
150 PRINT "At each number, type one of the items"
160 PRINT "to be sorted, then press RETURN."
170 PRINT
180 FOR K=1 TO MAX
190 PRINT K;" ";:INPUT NUM
200 N(K)=NUM
210 NEXT K
220 PRINT:PRINT "Sorting has begun."
230 REM *** BEGIN SORT ***
240 FOR J=1 TO MAX-1
250 FOR K=J+1 TO MAX
260 IF N(K)>N(J) THEN TEMP=N(J):N(J)=N(K):N(K)=TEMP
270 NEXT K
280 NEXT J
290 REM *** PRINT SORTED LIST ***
300 PRINT:PRINT "Do you want the sorted list printed"
310 PRINT:PRINT "   1 - smallest to largest, or"
320 PRINT "   2 - largest to smallest?"
330 PRINT:PRINT "WHICH (1 or 2) ";:INPUT WHICH
340 IF WHICH=2 THEN 370
350 FOR K=MAX TO 1 STEP-1
360 GOTO 380
370 FOR K=1 TO MAX
380 PRINT K, N(K)
390 NEXT K
```

The obvious question arises, "How do we sort strings?" We know that we can compare strings, just as we compare numbers; string A$ is "less than" string B$ if the strings are in the alphabetical order A$, B$. ATARI BASIC doesn't provide us with string arrays, so we have to use a technique slightly different from that of our previous sorting programs. Fortunately, an ATARI string may be as long as we want (subject to the amount of memory we have in our computer). This leads to what we like to call the "freight train."

Suppose we would like to sort this list of names: TAMI, JOHN, LISA, BILL, DEBI, RICK. To keep the discussion as simple as possible, we have chosen names with exactly four letters. Let's put the names together in one long string—this is the "freight train," with each name representing one of the "cars" of the train. Here's a diagram:

| TAMI | JOHN | LISA | BILL | DEBI | RICK |

FIGURE 15-3 "Freight Train" of Names

How long is this string? (The vertical separators don't count.) Right—24 characters: 6 names of 4 letters each. Now look at the following diagram (Figure 15-4), which shows the order of the cars in the freight train (1, 2, 3, 4, 5, or 6), the number of the *first* character in each car (below the car), and the number of the *last* character in each car (above the car).
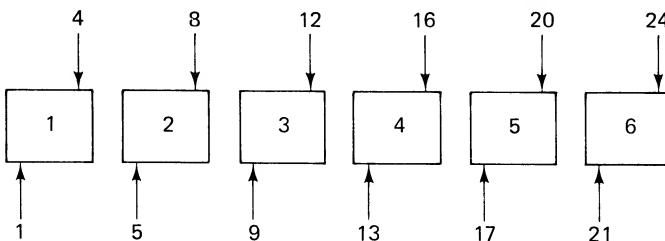
4	8	12	16	20	24
1	2	3	4	5	6
1	5	9	13	17	21

FIGURE 15-4 "Freight Train" Character Diagram

To be sure that you understand the diagram: The first car occupies characters 1 to 4 of the freight train; the second car, characters 5 to 8; the fifth car, characters 17 to 20; and the sixth car, characters 21 to 24.

Now a little detective work: Can you see a relationship between the number of the car and the number of the *last character* of the car? Here's the list:

| car 1 - 4 | car 3 - 12 | car 5 - 20 |
| car 2 - 8 | car 4 - 16 | car 6 - 24 |

Got it? Of course! The number of the last character is always exactly 4 times the number of the car because the cars have 4 characters each. Good. Now, how about the number of the *first character* in a car?

Well, let's look at the list for the first characters for each car:

car 1 - 1   car 3 - 9   car 5 - 17
car 2 - 5   car 4 - 13   car 6 - 21

Hmm. The number of the *first* character of each car is always just one more than the number of the *last* character of the *previous* car. Wow! Isn't that profound! Anyway, to get the number of the first character, do the following:

1. Subtract 1 from the number of the car

2. Multiply the result by 4

3. Add 1

These results are easier to write as formulas. Suppose that C represents the number of the car, then Table 15-3 gives the BASIC formulas for the numbers of the first and last characters of the car, and some examples.

TABLE 15-3   BASIC Formulas

Car Number	First Character	Last Character
C	4*(C-1)+1	4*C
3	4*(3-1)+1=9	4*3=12
1	4*(1-1)+1=1	4*1=4
6	4*(6-1)+1=21	4*6=24

If you're still with us, we're ready to see how to load the six names (remember our original problem?) into a single string, or freight train. Let the string be FT$. C will represent the number of each name, or car. The loading program is quite simple:

```
100 REM - Chapter 15, No. 11
105 REM - Load "Freight Train"
110 DIM FT$(24),NAME$(4)
120 GRAPHICS 0:PRINT
130 FOR C=1 TO 6
140 READ NAME$:PRINT NAME$
150 FIRST=4*(C-1)+1
160 LAST=4*C
170 FT$(FIRST,LAST)=NAME$
180 NEXT C
190 PRINT:PRINT FT$
200 DATA TAMI,JOHN,LISA,BILL,DEBI,RICK
```

We hope that your freight train didn't derail! Note that at lines 150 and 160 we calculated the numbers for the FIRST and LAST characters, using the formulas from Table 15-3. Line 170 is just the reverse of things we have done before—instead of taking part of a string out, we are putting part of a string in. We could have combined lines 150, 160, and 170 into one line:

```
150 FT$(4*(C-1)+1,4*C)=NAME$
```

but we think that the three lines may be easier to understand for beginners.

Just one more minor problem, and then we can do the sort program. Suppose you change the DATA slightly and run the previous program again:

```
200 DATA TAMI,JO,LISA,BILL,DEB,RICK
```

Did you get some garbage after the names JO and DEB? If the names are less than four letters long, we need to fill in with blanks; otherwise, the computer fills in with garbage. Add these two lines:

```
142 L=LEN(NAME$)
144 IF L<4 THEN FOR K=L+1 TO 4:NAME$(K,K)=" ":NEXT K
```

And now, finally, the program to sort the six names:

```
100 REM - Chapter 15, No. 12 - String Sort
110 DIM FT$(24),NAME$(4),TEMP$(4)
120 GRAPHICS 0:PRINT
130 FOR C=1 TO 6
140 READ NAME$
142 L=LEN(NAME$)
144 IF L<4 THEN FOR K=L+1 TO 4:NAME$(K,K)=" ":NEXT K
150 FIRST=4*(C-1)+1
160 LAST=4*C
170 FT$(FIRST,LAST)=NAME$
180 NEXT C
190 REM *** NAMES ARE LOADED IN "FREIGHT TRAIN" ***
200 REM *** BEGIN SORT ***
210 FOR J=1 TO 5
220 FOR K=J+1 TO 6
230 F1=4*(J-1)+1:L1=4*J
240 F2=4*(K-1)+1:L2=4*K
250 IF FT$(F2,L2)<FT$(F1,L1) THEN TEMP$=FT$(F1,L1):
FT$(F1,L1)=FT$(F2,L2):FT$(F2,L2)=TEMP$
260 NEXT K
270 NEXT J
280 REM *** PRINT SORTED LIST ***
290 FOR K=1 TO 6
300 FIRST=4*(K-1)+1:LAST=4*K
310 PRINT FT$(FIRST,LAST)
320 NEXT K
330 DATA TAMI,JOHN,LISA,BILL,DEBI,RICK
```

Can you see that we used the algorithm for the replacement sort (lines 210 to 270)? The J loop represents the first name in the pair to be compared, and the K loop represents the second name of the pair. At lines 230 and 240, we compute the numbers for the first and last characters in the first name (F1 and L1) and for the second name (F2 and L2) in the pair being compared. CHALLENGE: Fix the program to sort ten names.

EXERCISES

1. Given that   A(1)=8   B(1)=3.7   C(1)=4   T=1
                   A(2)=6   B(2)=9.2   C(2)=4   J=2
                   A(3)=10   K=3
                   A(4)=13   X=4,

    find the value of each of the following variables:

    a. A(2) _____   **e.** C(T) _____   **i.** C(X–3*T) _____

    b. A(K) _____   **f.** A(X) _____   **j.** A(J+1) _____

    c. A(T) _____   **g.** A(T+1) _____   **k.** A(X–K+J) _____

    d. B(J) _____   **h.** B(K–J) _____   **l.** A(J*K–X) _____

2. For each of the following programs, fill in the values for the sub-
    scripted variables.

    ```
    100 DIM P(4)          P(1) = _____       P(2) = _____
    110 FOR N=1 TO 4
    120 P(N)=2*N-1        P(3) = _____       P(4) = _____
    130 NEXT N
    ```

    ```
    100 DIM F(6)          F(2) = _____       F(3) = _____
    110 LET F(1)=1
    120 FOR K=2 TO 6      F(4) = _____       F(5) = _____
    130 F(K)=K*F(K-1)
    140 NEXT K            F(6) = _____
    ```

3. Correct the following program to print out the average of 8 numbers:

    ```
    100 DIM X(20)
    110 LET K+1=K
    120 READ X
    130 X=B(K)
    140 S=S+B(X)
    150 IF X=9999 THEN 170
    160 GOTO 100
    170 A=S/X
    180 PRINT "The average of the set is";A
    190 DATA 20,5,23,4,54,65,73,3,9999
    ```

4. The following program tells if you have won a computer in the big
drawing. Complete the sample RUNs. See printout.

```
100 REM - EXERCISES 15-2
110 DIM WIN(10)
120 FOR K=1 TO 10:READ W:WIN(K)=W:NEXT K
130 DATA  209,587,753,391,229,938,823,10,540,480
140 GRAPHICS 0:PRINT
150 PRINT "       BIG COMPUTER DRAWING"
160 PRINT
170 PRINT " Type your ticket number (1-1000)"
180 PRINT "to see if you have won a computer."
190 INPUT N
200 IF N<1 OR N>1000 THEN PRINT:PRINT
"A number from 1 to 1000, please!":GOT
O 190
210 FOR K=1 TO 10
220 IF WIN(K)=N THEN PRINT :PRINT "You
 have won a computer!":END
230 NEXT K
240 PRINT :PRINT "Sorry.  Better luck next time."
```

RUN

```
     BIG COMPUTER DRAWING

 Type your ticket number (1-1000)
to see if you have won a computer.
?125
```

RUN

```
     BIG COMPUTER DRAWING

 Type your ticket number (1-1000)
to see if you have won a computer.
?823
```

RUN

```
     BIG COMPUTER DRAWING

 Type your ticket number (1-1000)
to see if you have won a computer.
?1167
```

----

5. The measure of an interior angle of a regular polygon with n sides is given by the formula

$$\text{Interior angle} = \frac{180(n-2)}{n}$$

For example, a square (regular polygon with four sides) has interior angles of 90 degrees because

$$180 \times (4-2) = 360, \text{ and } 360/4 = 90$$

A regular pentagon (five sides) has interior angles of 108 degrees because

$$180 \times (5-2) = 540, \text{ and } 540/5 = 108.$$

Write an original program that will compute the measures in degrees of the interior angles of regular polygons with 3, 4, 5,...,19,20 sides. (Round these values to the nearest tenth of a degree.)
Store these values in the subscripted variable ANGLE, so that

```
ANGLE(3) =   60
ANGLE(4) =   90
ANGLE(5) = 108, etc.
```

Once the values are stored, the program will respond to the user as the sample RUN below illustrates.

```
RUN

How many sides ?3

In a regular polygon with 3 sides,
each interior angle measures
60 degrees.

READY
■
```

6. Write an original program that will allow the user to input 10 test scores.
The computer will print the average of the scores, the score that is closest to the average, and the score that is farthest from the average.
Store the scores in a subscripted variable, so that each one can be compared to the average. (You may wish to refer to program Chapter 15, No. 5.)

Here is a sample RUN:

```
RUN

Type in 10 test scores.
1 ?72
2 ?86
3 ?69
4 ?92
5 ?81
6 ?78
7 ?93
8 ?85
9 ?77
10 ?89

The average of the scores
is 82.2.

The score closest to the average
is 81.

The score farthest from the average
is 69.

READY
■
```

7. Think of a way to input people's birthdates in numerical form so that they could be sorted from oldest to youngest (or youngest to oldest) using the "universal" sort program "Chapter 15, No. 10." You do not need to write a program; just design a format for the input.

16

Variables With Two Subscripts (Matrices)

Suppose that we would like to keep track of the colors of all of the pixels in a GRAPHICS 3 display. Since we have 20 rows and 40 columns, we could use 20 subscripted variables with 40 subscripts each, storing the information for each row in one subscripted variable. That would be easier than using 20 × 40, or 800, separate variables, but still rather tedious. BASIC to the rescue again!

Double Subscripts

BASIC has *double-subscripted* variables to provide two-dimensional arrays. Such a two-dimensional array is called a *matrix* (plural matrices). Here is an example of a matrix with 3 rows and 4 columns. Such a matrix is referred to as a "3 by 4" matrix.

	column 1	*column 2*	*column 3*	*column 4*
row 1	3	0	1	2
row 2	1	3	1	0
row 3	2	1	3	3

Let's call a double-subscripted variable MAT. We'll give it 3 rows and 4 columns by using this DIMension statement:

```
110 DIM MAT(3,4)
```

When we refer to one of the elements of the matrix MAT, we name the *row first* and the *column second*. Here is our 4 × 3 matrix again, but this time we've inserted the names of the elements instead of their values.

	column 1	column 2	column 3	column 4
row 1	MAT(1,1)	MAT(1,2)	MAT(1,3)	MAT(1,4)
row 2	MAT(2,1)	MAT(2,2)	MAT(2,3)	MAT(2,4)
row 3	MAT(3,1)	MAT(3,2)	MAT(3,3)	MAT(3,4)

Comparing our two matrices, we see that MAT(2,1)=1, MAT(1,2)=0, MAT(3,4)=3, and MAT(1,4)=2. Just as in the case of arrays (single-subscripted variables), we get some freebies with matrices. There is a 0 row and a 0 column, so our matrix MAT actually contains these additional elements that we didn't show in the chart:

```
MAT(0.0), MAT(0,1), MAT(0,2), MAT(0,3), MAT(0,4)
MAT(1,0)
MAT(2,0)
MAT(3,0)
```

Thus our 3 × 4 matrix really contains 4 rows and 5 columns, or a total of 20 (4 × 5) elements.

Initializing a Matrix

To initialize an array, we use a single FOR...NEXT loop. The two-dimensional counterpart is a pair of nested FOR...NEXT loops. Here is a program to set all of the elements (including those in the 0 row and 0 column) of our 3 × 4 matrix equal to 0:

```
100 REM - Chapter 16, No.1
110 DIM MAT(3,4)
120 REM *** INITIALIZE MATRIX MAT ***
130 FOR ROW=0 TO 3
140 FOR COLUMN=0 TO 4
150 MAT(ROW,COLUMN)=0
160 NEXT COLUMN
170 NEXT ROW
```

We can use this standard program, or *routine*, to initialize any matrix; all we need to do is change some numbers at lines 110, 130, and 140. For example, to initialize a matrix with 12 rows and 9 columns, we would change these lines:

```
110 DIM MAT(12,9)
130 FOR ROW=0 TO 12
140 FOR COLUMN=0 TO 9
```

Suppose that we would like to create the matrix in the first chart on the previous page. We could read the individual items from DATA statements. Add these statements to the previous program to initialize the matrix:

```
100 REM - Addition to Chapter 16, No.1
200 REM *** FILL IN THE MATRIX ***
210 FOR ROW=1 TO 3
220 FOR COLUMN=1 TO 4
230 READ NUM
240 MAT(ROW,COLUMN)=NUM
250 NEXT COLUMN
260 NEXT ROW
270 DATA 3,0,1,2
280 DATA 1,3,1,0
290 DATA 2,1,3,3
```

Notice that this is almost the same as the program to initialize the matrix. At lines 230 and 240, a number is read from a DATA statement and then stored in the matrix; we can't read numbers directly into a matrix.

Now let's add some lines to plot the points that correspond to the locations in the matrix, using the *values* in the matrix as color values.

```
100 REM - Addition to Chapter 16, No.1
300 REM *** PLOT POINTS ***
310 GRAPHICS 3
320 FOR ROW=1 TO 3
330 FOR COLUMN=1 TO 4
340 C=MAT(ROW,COLUMN):COLOR C
350 PLOT COLUMN,ROW
360 NEXT COLUMN
370 NEXT ROW
```

You should get a pattern in the upper left corner of the screen with the following colors:

	column 1	*column 2*	*column 3*	*column 4*
row 1	blue	black	orange	green
row 2	orange	blue	orange	black
row 3	green	orange	blue	blue

Now let's do something just a bit more ambitious. This time we'll fill in a 10 × 10 square in the upper left corner of the screen. You'll need to make these additions and/or changes:

```
100 REM - Addition to Chapter 16, No.1
110 DIM MAT(10,10)
130 FOR ROW=0 TO 10
140 FOR COLUMN=0 TO 10
210 FOR ROW=1 TO 10
220 FOR COLUMN=1 TO 10
270 DATA 1,1,1,1,1,1,1,1,1,1
272 DATA 1,0,0,0,0,0,0,0,0,1
274 DATA 1,0,2,2,2,2,2,2,0,1
276 DATA 1,0,2,0,0,0,0,2,0,1
278 DATA 1,0,2,0,3,3,0,2,0,1
280 DATA 1,0,2,0,3,3,0,2,0,1
282 DATA 1,0,2,0,0,0,0,2,0,1
284 DATA 1,0,2,2,2,2,2,2,0,1
286 DATA 1,0,0,0,0,0,0,0,0,1
290 DATA 1,1,1,1,1,1,1,1,1,1
320 FOR ROW=1 TO 10
330 FOR COLUMN=1 TO 10
```

Run the program again once or twice and study the listing. When you're quite certain that you understand all the details, make these additions:

```
100 REM - Addition to Chapter 16, No. 1
310 GRAPHICS 3+16
312 X=INT(31*RND(0))-1
314 Y=INT(15*RND(0))-1
350 PLOT X+COLUMN,Y+ROW
355 SOUND 1,3*C*(X+Y+2),10,8
356 SOUND 2,C*(X+Y+2),10,8
380 GOTO 312
```

You may want to experiment with the sound statements at lines 355 and 356. Do you see that lines 312 and 314 produce random numbers for coordinates of the upper left corner of the square? Try adding some SETCOLOR statements to make the design the colors you want, or change the design. The possibilities are endless!

Using Matrices - Tallying

Another use of matrices in BASIC is to tally, or *count*, items that fall into various categories. Suppose that we have identified the students in grades 9 to 12 who can write simple BASIC programs, and we would now like to see how many boys and how

many girls there are at each grade level. We'll collect a card like
the sample below from each person:

```
                I speak BASIC.

      ┌──────────┐        ┌──────────┐
      │          │        │          │
      │    10    │        │     2    │
      │          │        │          │
      └──────────┘        └──────────┘
         GRADE               SEX
                           (1=BOY )
                           (2=GIRL)
```
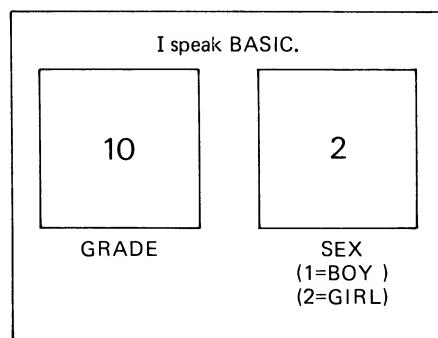
FIGURE 16-1   Sample Tallying Card

Then we'll put the pair of numbers from each card in a data
statement and store the results in a matrix. The card above
represents a 10th grade girl, so we'll put the numbers 10 and 2 in
a data statement to register her ability to program in BASIC.
We'll need a matrix with 4 rows (for grades 9, 10, 11, and 12) and 2
columns (for boys and girls). As we read each pair of numbers
from the data statement, we'll add one to the value in the appro-
priate place in the matrix. Now try the program.

```
100 REM - Chapter 16, No.2
110 DIM T(4,2)
120 REM *** INITIALIZE MATRIX T ***
130 FOR ROW=0 TO 4
140 FOR COL=0 TO 2
150 T(ROW,COL)=0
160 NEXT COL
170 NEXT ROW
180 REM *** READ & RECORD "VOTES" ***
190 COUNT=0
200 READ GRADE,SEX
210 IF GRADE=-1 THEN 250
220 T(GRADE-8,SEX)=T(GRADE-8,SEX)+1
230 COUNT=COUNT+1
240 GOTO 200
250 REM *** TOTAL ROWS ***
260 FOR ROW=1 TO 4
270 T(ROW,0)=T(ROW,1)+T(ROW,2)
275 T(0,0)=T(0,0)+T(ROW,0)
280 NEXT ROW
290 REM *** TOTAL COLUMNS ***
300 FOR COL=1 TO 2
310 T(0,COL)=T(1,COL)+T(2,COL)+T(3,COL)+T(4,COL)
320 NEXT COL
330 REM *** PRINT RESULTS ***
340 GRAPHICS 0:PRINT
350 PRINT "NUMBER OF STUDENTS BY GRADE AND SEX"
360 PRINT "  WHO CAN WRITE A SIMPLE PROGRAM"
```

```
370 PRINT "        IN THE BASIC LANGUAGE"
380 PRINT
390 PRINT "GRADE","BOYS","GIRLS","TOTAL"
400 PRINT "=====","====","=====","====="
405 PRINT
410 FOR ROW=1 TO 4
420 PRINT " ";ROW+8," ";T(ROW,1)," ";T(ROW,2)," ";T(ROW,0)
430 PRINT
435 NEXT ROW
436 PRINT
440 PRINT "TOTAL"," ";T(0,1)," ";T(0,2)," ";T(0,0)
450 PRINT:PRINT:PRINT
500 DATA  10,2,9,1,9,1,11,1,12,1,10,1,10,2,10,1,12,2,9,2
510 DATA 11,1,9,1,11,1,12,1,10,2,11,2,12,1,9,2,9,1,12,2
520 REM *** -1,-1 ARE FLAGS ***
530 DATA -1,-1
```

NOTE: The variable COUNT counts the number of responses.

Let's consider a few features of the preceding program. At line 210 we test for the flag (-1) to see if all the data has been read. If it has, we tally the rows and then the columns. To understand line 220, it might be well to see exactly what our matrix T looks like. In Table 16-1, we have indicated what each box in the matrix represents.

TABLE 16-1   Matrix for Student Programmers

		column 0	*column 1*	*column 2*
row 0		GRAND TOTAL	BOYS TOTAL	GIRLS TOTAL
row 1		9th TOTAL	9th BOY	9th GIRL
row 2		10th TOTAL	10th BOY	10th GIRL
row 3		11th TOTAL	11th BOY	11th GIRL
row 4		12th TOTAL	12th BOY	12th GIRL

Note that the number of each row is 8 *less than the grade* that that row represents. Thus, at line 220, we refer to **T(GRADE-8,SEX)**. For example, if we read **10,2** from the DATA statement, then we add one to **T(2,2)**. Thus, line 220 records the individual "votes" in the proper place. Lines 260 to 280 do two things. First, we find the sum of the number of boys and the number of girls for each grade and store the result in column 0 for that grade (line 270). Second, we add that total to the number in **T(0,0)**, the grand total. You can see that the zero rows and columns are handy for storing totals. Lines 300 to 320 find the totals for the boys and for the girls and store them in **T(0,1)** and **T(0,2)**. The printing part of the program should be easy to understand.

You could modify the program to tally the votes in an election—let the columns represent the candidates and let the

rows represent the various grade levels. Here is an example (Table 16-2) of an appropriate matrix:

TABLE 16-2 Matrix to Tally Election Votes

	column 0	candidate 1	candidate 2
row 0	TOTAL VOTES	#1 TOTAL	#2 TOTAL
row 1	9th TOTAL	#1 9th	#2 9th
row 2	10th TOTAL	#1 10th	#2 10th
row 3	11th TOTAL	#1 11th	#2 11th
row 4	12th TOTAL	#1 12th	#2 12th

Make a tallying program to use for your next school election!

EXERCISES

1. The statement

   1. _____

   ```
   110 DIM GRID(6,4)
   ```

   sets up a matrix with how many elements?

2. The process of filling all of the elements of a matrix with values is called _?_ the matrix.

   2. _____

3. How many rows are there in the matrix defined by the statement

   3. _____

   ```
   110 DIM RECT(5,2)    ?
   ```

4. Trace the following program without using the computer and then complete the questions.

   ```
   100 REM - EXERCISES 16-1
   110 DIM HUE(3,4)
   120 FOR ROW=1 TO 3
   130 FOR COL=1 TO 4
   140 READ NUM
   ```

```
150 HUE(ROW,COL)=NUM
160 NEXT COL
170 NEXT ROW
180 DATA 1,1,1,1,2,2
190 DATA 2,2,3,3,3,3
```

HUE(1,1) = _____   HUE(2,3) = _____   HUE(3,2) = _____

HUE(2,2) = _____   HUE(2,4) = _____   HUE(1,4) = _____

5. Describe the output if the following statements are added to the program in problem 4 above.

```
200 GRAPHICS 3
210 FOR ROW=1 TO 3
220 FOR COL=1 TO 4
230 COLOR HUE(ROW,COL)
240 PLOT COL+10,ROW+5
250 NEXT COL
260 NEXT ROW
```

_____

_____

6. Fred, Chip, Suzy, and Tina compared their scores at STAR RATS (a video game) for the months of September, October, November, December, and January. They made the following chart. Write a program that will use the given data lines to compute the totals and print out the chart.

      Hints: **a.** use a matrix to store the data.

             b. Compute the monthly and personal totals and store them in the zero rows and columns of the matrix.

	SEP	OCT	NOV	DEC	JAN	TOTAL
Fred	1125	1375	1200	1525	1475	?
Chip	1325	1475	1650	1375	1525	?
Suzy	950	1175	1025	1450	1375	?
Tina	1275	1475	1375	1675	1725	?
TOTAL	?	?	?	?	?	

Here are the data lines to use:

```
200 DATA Fred,1125,1375,1200,1525,1475
210 DATA Chip,1325,1475,1650,1375,1525
220 DATA Suzy,950,1175,1025,1450,1375
230 DATA Tina,1275,1475,1375,1675,1725
```

7. Val Vexel, an American who lives in Frankfurt, West Germany, monitored the Deutsche Mark – US dollar exchange rate for four weeks and made this chart:

	MON	TUE	WED	THU	FRI
Week 1	2.36	2.35	2.34	2.33	2.32
Week 2	2.31	2.34	2.36	2.39	2.37
Week 3	2.35	2.35	2.36	2.33	2.36
Week 4	2.37	2.38	2.40	2.40	2.41

NOTE: 2.36 means $1 will buy 2.36 Deutsche Marks.

Write a program that will use the given data lines to print out the highest, lowest, and average exchange rate for each week. Store the data in a matrix, as in the previous problem. Here are the data lines to use:

```
200 DATA 2.36,2.35,2.34,2.33,2.32
210 DATA 2.31,2.34,2.36,2.39,2.37
220 DATA 2.35,2.35,2.36,2.33,2.36
230 DATA 2.37,2.38,2.40,2.40,2.41
```

17

Subroutines

It often happens in a program that we would like to do the same thing at various places in the program. It might be quite tedious to duplicate the program instructions several times. Fortunately, we have a BASIC command that lets us avoid this duplication—the GOSUB statement.

Back in Chapter 3, we discussed a music program (Chapter 3, No. 7) that included the GOSUB statement (without discussion). We repeat the program here:

```
100 SOUND 0,121,10,10:GOSUB 300
110 SOUND 0,121,10,10:GOSUB 300
120 SOUND 0,96,10,10:GOSUB 300
130 SOUND 0,96,10,10:GOSUB 300
140 SOUND 0,81,10,10:GOSUB 300
150 SOUND 0,81,10,10:GOSUB 300
160 SOUND 0,96,10,10:GOSUB 300
170 STOP
300 FOR HOLD=1 TO 100:NEXT HOLD
310 FOR OFF=1 TO 10
320 SOUND 0,0,0,0
330 NEXT OFF
340 RETURN
```

After each sound statement we needed to do two things:

1. **Hold the note for a certain length of time (in this case, a count of 100).**

2. **Turn the note off for a certain length of time (a count of 10).**

This was done with the GOSUB statement: GOSUB 300 means

"GO TO LINE 300 AND EXECUTE ALL STATEMENTS FROM LINE 300 UNTIL A 'RETURN' STATEMENT (LINE 340) IS ENCOUNTERED. THEN GO BACK TO THE NEXT STATE-MENT AFTER THE GOSUB STATEMENT."

Thus, a GOSUB statement is *always* used in conjunction with a RETURN statement. The program statements beginning with the line named in the GOSUB statement and ending with the RETURN statement are known as a *subroutine*. For example, in the program above, the following lines are a sub-routine:

```
300 FOR HOLD=1 TO 100:NEXT HOLD
310 FOR OFF=1 TO 10
320 SOUND 0,0,0,0
330 NEXT OFF
340 RETURN
```

Instead of repeating lines 300 to 330 seven times in the program (once for each SOUND statement), we need to write them only once as a subroutine.

We can use the same technique in a graphics program if we want to hold a pattern on the screen for a certain length of time before changing it or adding to it. Try this program:

```
100 REM - Chapter 17, No. 1
110 GRAPHICS 3+16
120 COLOR 1
130 X=INT(40*RND(0))
140 Y=INT(24*RND(0))
150 IF X/2=INT(X/2) THEN GOSUB 300:GOTO 120
160 PLOT X,Y:SOUND 0,5*X,10,6:FOR W=1 TO 5:NEXT W
170 GOTO 120
300 SETCOLOR 0,4,4
310 FOR W=1 TO 30:SOUND 0,60,2,10:NEXT W
320 SOUND 0,0,0,0
330 SETCOLOR 0,14,10
340 RETURN
```

Notice that line 150 prevents points with even X-coordinates from being plotted. Instead, the points already plotted flash red, and a buzzer sounds briefly.

You may be familiar with Euclid's Algorithm from your previous math classes. It is a procedure for finding the greatest common factor (G.C.F.) of two integers. The procedure can be summarized as follows:

1. **Divide the larger integer by the smaller.**
2. **Divide the divisor by the remainder.**
3. **Repeat 2. until the remainder is zero (the division "comes out even"). The last divisor is the G.C.F. of the two integers.**

As an example, the procedure for finding the G.C.F. of 20 and 12 is shown below:

$$
1)\quad 12\overline{)\underset{\underline{12}}{20}}^{\,1} \qquad 2)\quad 8\overline{)\underset{\underline{8}}{12}}^{\,1} \qquad 3)\quad 4\overline{)\underset{\underline{8}}{8}}^{\,2}
$$

$$
\begin{array}{ccc}
8 & 4 & 0
\end{array}
$$

Because the last remainder is zero, the G.C.F. of 20 and 12 is 4 (the last divisor).

To reduce a fraction, we divide the numerator and denominator by their G.C.F. For example,

$$\frac{12}{20} \text{ reduces to } \frac{12/4}{20/4} \text{ or } \frac{3}{5}$$

Here is a program that will reduce a fraction using Euclid's Algorithm. The Algorithm is written as a subroutine starting at line 12000. We can save the subroutine on disk and call it up whenever we need it. Over a period of time we can build a library of subroutines to use in our programs.

```
100 REM - Chapter 17, No. 2
110 GRAPHICS 0:PRINT
120 PRINT "REDUCING FRACTIONS"
130 PRINT
140 PRINT "Type the numerator of a fraction."
150 INPUT NUM
160 PRINT
170 PRINT "Type the denominator of a fraction."
180 INPUT DEN
190 N99=NUM:D99=DEN
200 GOSUB 12000
210 GRAPHICS 2
220 POSITION 2,4
230 PRINT #6;NUM;"/";DEN;" = ";NUM/GCF;"/";DEN/GCF
240 END
```

```
12000 REM *****************************
12001 REM **                         **
12002 REM **   EUCLID'S ALGORITHM    **
12003 REM **   N99 = LARGER NUMBER   **
12004 REM **   D99 = SMALLER NUMBER  **
12005 REM **   GCF = G.C.F.          **
12006 REM **                         **
12007 REM *****************************
12008 IF D99>N99 THEN T99=D99:D99=N99:N99=T99
12010 Q99=INT(N99/D99)
12020 R99=N99-Q99*D99
12030 IF R99=0 THEN GCF=D99:GOTO 12070
12040 N99=D99
12050 D99=R99
12060 GOTO 12010
12070 RETURN
```

Notice that in order to use the subroutine, the numerator must be placed in memory location N99 and the denominator must be placed in memory location D99. In a subroutine that will be used frequently, it is a good idea to use names (for the variables) that will probably not be used in the main program.

Notice also that we must have an END statement at line 240 to prevent entering the subroutine without a GOSUB statement. If this should occur, an error number 16 would result.

Subroutines in Music

In the section on the GOSUB statement, we repeated a music program that used subroutines. One subroutine was used to hold the note for a count of 100, and the other was used to turn the note off for a count of ten. Following is a program that does essentially the same thing, but it allows for the possibility of notes of different time values (quarter notes, eighth notes, etc.) and for turning a note off for a shorter count to tie it to the next note (slur). The numerical values for the notes are read in from DATA statements. For each note we need to read three values: the pitch (see Appendix 2), the time value of the note (see chart following), and a number to indicate whether or not the note is tied to the next one (1 = tie, 10 = no tie). Here is a chart that shows the numbers to use for various time values:

64th note	64
32nd note	32
16th note	16
eighth note	8
quarter note	4
half note	2
whole note	1
dotted note	8., 16., 2., etc.

Notice that we give the subroutine to hold a note the name HOLDNOTE at line 140. Then at line 190 we can say GOSUB HOLDNOTE instead of GOSUB 300. If we need to refer to a subroutine several times in a program, it might be easier to refer to it by name than by number. Also study the procedure for dealing with a dotted note at lines 160 to 176.

```
100 REM - Chapter 17, No. 3
105 DIM TIMEVAL$(3)
110 GRAPHICS 0:PRINT
120 PRINT "CHOOSE A TEMPO"
121 PRINT
122 PRINT "   1 = FAST"
123 PRINT "   2 = MEDIUM FAST"
124 PRINT "   3 = NORMAL"
125 PRINT "   4 = MEDIUM SLOW"
126 PRINT "   5 = SLOW"
127 PRINT
128 PRINT "YOUR CHOICE ";
130 INPUT TEMPO
140 HOLDNOTE=300
150 OFFNOTE=400
159 REM *** LOOP TO READ NOTES ***
160 READ PITCH,TIMEVAL$,TIE
170 IF PITCH<0 THEN END
172 L=LEN(TIMEVAL$)
174 IF TIMEVAL$(L,L)="." THEN TIMEVAL=
0.75*VAL(TIMEVAL$(1,L-1)):GOTO 180
176 TIMEVAL=VAL(TIMEVAL$)
180 SOUND 0,PITCH,10,10
190 GOSUB HOLDNOTE
200 GOSUB OFFNOTE
210 GOTO 160
299 REM *** SUBROUTINE "HOLDNOTE"
300 FOR HOLD=1 TO 2*TEMPO*64/TIMEVAL
310 NEXT HOLD
320 RETURN
399 REM *** SUBROUTINE "OFFNOTE"
400 FOR OFF=1 TO TIE/2
410 SOUND 0,0,0,0
```

```
420 NEXT OFF
430 RETURN
499 REM *** DATA FOR "GAUDEAMUS" ***
501 DATA 60,8.,10,81,16,10,81,4,10,60,4,10
502 DATA 72,8.,10,72,16,10,72,2,10
503 DATA 64,8.,10,60,16,10,53,4,10,64,4,10
504 DATA 60,8,10,47,8,10,60,2,10
505 DATA 60,8.,10,81,16,10,81,4,10,60,4,10
506 DATA 72,8.,10,72,16,10,72,2,10
507 DATA 64,8.,10,60,16,10,53,4,10,64,4,10
508 DATA 60,8,10,47,8,10,60,2,10
509 DATA 129,8.,10,121,16,10,108,4,10,108,4,10
510 DATA 96,8,10,121,8,10,108,4,10,108,4,10
511 DATA 129,8.,10,121,16,10,108,4,10,108,4,10
512 DATA 96,8,10,121,8,10,108,4,10,108,4,10
513 DATA 121,8.,10,129,4,10,0,16,10,91,8,10,108,8,10
514 DATA 121,4,10,129,4,10,121,4,10
515 DATA 121,8.,10,129,4,10,0,16,10,91,8,10,108,8,10
516 DATA 96,4,10,108,4,10,121,4,10
999 DATA -1,0,0
```

Each DATA line of the program represents one measure of music. Thus our song is sixteen measures long. Keeping the measures separate makes it easier to find errors when you are debugging the program. You can add an interesting effect to the song by adding this line:

```
185 SOUND 1,PITCH+1,10,6
```

Add this line for a very sour song:

```
186 SOUND 2,PITCH+9,10,6
```

A Library of Subroutines

We have suggested the idea of building a library of subroutines for use in various programs. The example we gave was a program, used to reduce a fraction; it included a subroutine for finding the greatest common factor (G.C.F.) of two numbers by using Euclid's Algorithm. Suppose we type in lines 12000 to 12070 (the subroutine) of the program, and then save it on disk using the following command:

```
LIST "D:EUCLID.LST"
```

The LIST command saves the program on disk in a slightly different way from the SAVE command. Recall that if we LOAD a program from a disk, then it will erase any program that was already in the computer memory. However, if we have saved a program with the LIST command (as above), then we can load it into the computer with the command

```
ENTER "D:EUCLID.LST"
```

and any program already in memory will *not be erased*. Thus we can tack-on a subroutine to any existing program. Then we can SAVE the entire program in the normal way. Note that we added the extender ".LST" to our subroutine name to indicate that it was a LISTed program instead of a SAVEd program and that to recall it from disk we need the ENTER command instead of the LOAD command.

We'll review this entire procedure with an example. The least common multiple (L.C.M.) of two numbers can be found by dividing their product by their greatest common factor (G.C.F.). For example, the L.C.M. of 12 and 18 is equal to their product (216) divided by their G.C.F. (6), or 36. Thus, the smallest number that is divisible by both 12 and 18 is 36. Now suppose that you have LISTed the program for Euclid's Algorithm as indicated above, using the same EUCLID.LST. Follow these steps:

1. **Type this program:**

```
100 REM - Chapter 17, No. 4
110 EUCLID=12000
120 GRAPHICS 0:PRINT
130 PRINT "LEAST COMMON MULTIPLE"
140 PRINT
150 PRINT "Type two integers."
160 PRINT "Example:   12,18"
170 INPUT A,B
180 N99=A
190 D99=B
200 GOSUB EUCLID
210 LCM=A*B/GCF
220 PRINT
230 PRINT "The L.C.M. of ";A;" and ";B;" is ";LCM;"."
240 END
```

2. **Recall the subroutine for Euclid's Algorithm with the command**

```
ENTER "D:EUCLID.LST"
```

3. **Test the program with the following pairs of numbers. The correct result is shown in parentheses.**

$$12,18 \quad (\text{L.C.M.} = 36)$$
$$18,12 \quad (\text{L.C.M.} = 36)$$
$$5,17 \quad (\text{L.C.M.} = 85)$$
$$13,13 \quad (\text{L.C.M.} = 13)$$

4. **If you get the correct results for the previous tests, then save the complete program (including the subroutine) on disk using this command:**

```
SAVE "D:LCM"
```

We have used the subroutine EUCLID in two different programs: first, to reduce a fraction, and second, to find the L.C.M. of two numbers. Both the programs used the subroutine to find the G.C.F. of two numbers. In order to use the subroutine, we had to store the two numbers in N99 and D99. The REMark statements at the beginning of a subroutine should indicate what variable names we have to use to enter the subroutine and what variable name(s) are used for the outcome. Look again at the REMark statements for EUCLID:

```
12000 REM  ******************************
12001 REM  **                          **
12002 REM  **   EUCLID'S ALGORITHM      **
12003 REM  **   N99 = LARGER NUMBER     **
12004 REM  **   D99 = SMALLER NUMBER    **
12005 REM  **   GCF = G.C.F.            **
12006 REM  **                          **
12007 REM  ******************************
```

We see that N99 refers to the larger number and D99 refers to the smaller number. (If they happen to be in the opposite order, line 12008 will switch them.) The result of the subroutine, the G.C.F., is stored in GCF. As you work with your own subroutines, you will see that such a system of REMark statements is very useful (if not absolutely necessary!) for keeping track of how to enter the subroutines. Because you may want to use more than one subroutine in a program, you should give each subroutine a unique set of line numbers—numbers that you do not use in the main program.

Here are some sample subroutines that you might find useful in your programs:

1. **Find the number of digits in a number. This might be useful for spacing purposes in screen output, particularly if you are using graphics modes 1 or 2.**

```
15000 REM  ******************************
15001 REM  **                          **
15002 REM  ** DIGITS FUNCTION          **
15003 REM  ** A99  = NUMBER            **
15004 REM  ** NDIG = NUMBER OF DIGITS  **
15005 REM  **          IN A99          **
15006 REM  **                          **
15007 REM  ******************************
15010 NDIG=INT(CLOG(A99))+1
15020 RETURN
```

2. Find the X- and Y-coordinates of the points on a circle and store them in arrays.

```
16000 REM ***************************
16001 REM **                       **
16002 REM ** CIRCLE COORDINATES    **
16003 REM ** XCIR(A99),YCIR(A99) = **
16004 REM **    COORDINATES OF PT. **
16005 REM **    ON UNIT CIRCLE, AT **
16006 REM **    ANGLE A99 DEGREES. **
16007 REM **                       **
16008 REM ***************************
16010 IF SW99=0 THEN DIM XCIR(360),YCIR(360)
16020 DEG :PRINT "COUNTING 0 TO 360..."
16030 FOR A99=0 TO 360
16040 X99=COS(A99)
16050 Y99=SIN(A99)
16060 XCIR(A99)=X99
16070 YCIR(A99)=Y99 :PRINT A99
16080 NEXT A99
16090 SW99=SW99+1
16100 RETURN
```

3. Draw a circle using the coordinates found by the previous subroutine.

```
17000 REM ************************
17001 REM **                    **
17002 REM **   DRAW CIRCLE      **
17003 REM **   H99,K99 = CENTER **
17004 REM **   R99     = RADIUS **
17005 REM **   C99     = COLOR  **
17006 REM **   S99     = STEP   **
17007 REM **                    **
17008 REM ************************
17010 DEG :COLOR C99
17020 FOR A99=0 TO 360 STEP S99
17030 TRAP 17080
17040 X99=XCIR(A99):Y99=YCIR(A99)
17050 X99=INT(1.2*R99*X99+0.5)
17060 Y99=INT(R99*Y99+0.5)
17070 PLOT X99+H99,K99-Y99:GOTO 17090
17080 TRAP 40000
17090 NEXT A99
17100 RETURN
```

Here is a program to draw a circle using the previous two subroutines. Change lines 130 to 170 to change the center,

radius, color, and number of points plotted (S99 is the number of degrees between points). Be sure to add subroutines 2. and 3. from page 239 before you run the program!

```
100 REM - Chapter 17 No. 5
110 GOSUB 16000
120 GRAPHICS 7
130 H99=80
140 K99=40
150 R99=20
160 C99=1
170 S99=5
180 GOSUB 17000
190 END
```

Here are some variations to the program above that you might enjoy. Start with the program as written above, and then make the additions or corrections in order from 1. to 5. To run the program after a modification, type GOTO 120 and press RE-TURN. (The circle data is already stored.)

1. Draws circle, then erases it.

```
170 S99=30
185 C99=1-C99:GOTO 180
```

2. Draws a whole row of circles.

```
130 FOR H99=0 TO 159 STEP 10
170 S99=5
185 NEXT H99
```

3. Similar to 2., but on a diagonal.

```
140K99=H99/2
150 R99=20
```

4. Changes the circles to ovals.

```
17065 X99=2*X99
```

5. Draws circles with different colors, radii, and centers.

```
130 H99=INT(160*RND(0))
```

```
140 K99=INT(80*RND(0))
150 R99=5*INT(10*RND(0)+1)
160 C99=INT(3*RND(0)+1)
170 S99=5
185 GOTO 130
```

Note that when we use the previous program and subroutines to draw circles, we need to use subroutine "16000" only once. After the coordinates of the points on the circle are calculated and stored in the arrays, we just need to read them out again to plot another circle.

Some Reminders About Using Subroutines

1. You may enter a subroutine only with a GOSUB statement. Therefore, be sure that there is an END statement in the program *before* the subroutine(s) begin. Refer to line 240 in program Chapter 17, No. 1 for an example.

2. Be sure to move the values that will be used in the subroutine into the proper variables to enter the subroutine. For example, in the program Chapter 17, No. 2, the numerator and denominator of the fraction were INPUT into NUM and DEN and then transferred at line 190 into N99 and D99, the variables used in the subroutine.

3. A subroutine must end with a RETURN statement. After a subroutine is executed, control passes to the *next statement* after the GOSUB that initiated the subroutine. The next statement may be in the *same line* as the GOSUB statement. Here is a very simple example:

```
100 REM - Chapter 17, No. 6
110 GRAPHICS 0:PRINT
120 DIM A$(3)
130 PRINT "Do you understand subroutines"
140 PRINT "(Type YES or NO)"
150 INPUT A$
160 IF A$="YES" THEN GOSUB 500:GOSUB 600:GOSUB 700:END
170 PRINT: PRINT "YOU NEED HELP!":END
500 REM *** TURN SOUND ON ***
510 SOUND 0,60,10,10
520 RETURN
600 REM *** COUNT TO 100 ***
610 FOR K99=1 TO 100
620 NEXT K99
630 RETURN
700 REM *** TURN SOUND OFF ***
710 SOUND 0,0,0,0
720 RETURN
```

EXERCISES

1. A GOSUB statement is always used in conjunc-   1. _____

   tion with a __?__

2. A subroutine is __?__   2. _____

   a. An underwater naval maneuver.

   b. A short program that is part of another
      program.

   c. An optional part of a program that con-
      tains instructions to the program user.

3. Which *one* of the following statements is true?   3. _____

   a. A subroutine that ends with a RETURN
      statement may be entered with a GOTO
      statement.

   b. 120 GOSUB 500:GOSUB 500:GOSUB 500
      is a valid statement.

   c. The last statement before a subroutine must
      be an END statement.

4. Write the printout of the following program:

```
100 FOR K=1 TO 7
110 IF K/2=INT(K/2) THEN GOSUB 200
120 IF K/5=INT(K/5) THEN GOSUB 300
130 PRINT K
140 NEXT K
150 END
200 PRINT "Go Navy!"
210 RETURN
300 PRINT "This is a subroutine."
310 RETURN
```

5. Write a subroutine that will find the position of a given string within a given larger string. Here is a sample RUN of a program that uses the subroutine:

```
RUN

Type the string to be searched.
?ATARI COMPUTERS ARE FUN.

Type the string to be found.
?PUT

"PUT" is in position (10,12) of
"ATARI COMPUTERS ARE FUN."
```

6. Modify program Chapter 17, No. 3 so that the user can INPUT the coordinates of the center, the radius, and the color (red, white, or blue). Use GRAPHICS 7 and let the step be 5. A sample RUN might show this display in the text window:

```
RUN

COORDINATES OF CENTER ?80,40

RADIUS ?35

COLOR ?RED
```

(The computer draws a red circle with center at **80,40** and radius 35 in GRAPHICS 7, then gives the user a prompt for another circle without clearing the screen.)

7. Write a program to add fractions that uses Euclid's Algorithm as a subroutine. The answer should be written in lowest terms but may be an improper fraction. Here is a sample RUN:

```
RUN

Numerator of 1st fraction   ?2
Denominator of 1st fraction ?3

Numerator of 2nd fraction   ?5
Denominator of 2nd fraction ?6

2/3 + 5/6 = 3/2
```

Review

You've come a long way! You should now be able to solve a variety of problems using ATARI BASIC. In this chapter, we'll review by considering two rather long sample programs. The Exercises at the end of the chapter will test your understanding of them.

The following program uses GRAPHICS 0 and some control graphics characters to produce a bar graph. The graph shows how many video games each of five students own. The names of the students and the number of their video games are stored in data statements.

The first step in designing such a program is to plan the display on a piece of graph paper. Figure 18-1 on page 246 shows how we'd like our screen display to look. Pay careful attention to the numbers of the rows and columns.

Keep in mind as we go through the various parts of the program that we are developing a program that we could modify for other data—if we just wanted a display for one set of data, then we could do the graph with 22 PRINT statements!

```
                    (read column numbers down)

       0123456789111111111122222222223333333333
                0123456789012345678901234567890

   0 ┌─────────────────────────────────────────┐
   1 │                                          │
   2 │         ┌─────────────────────────┐      │
   3 │  TED    │*********************     │      │
   4 │         │                         │      │
   5 │  TINA   │*************            │      │
   6 │         │                         │      │
   7 │  BARRY  │****************         │      │
   8 │         │                         │      │
   9 │  LISA   │*********               │      │
  10 │         │                         │      │
  11 │  KEVIN  │****************************│    │
  12 │         │                         │      │
  13 │         └─────────────────────────┘      │
  14 │         ││││││││││││││││││││││││││││       │
  15 │                                          │
  16 │         0    5   10   15   20   25   30  │
  17 │                                          │
  18 │               Number of Games            │
  19 │                                          │
  20 │         =================================│
  21 │         VIDEO GAMES OWNED BY CERTAIN STUDENTS │
  22 │         =================================│
  23 └─────────────────────────────────────────┘
```
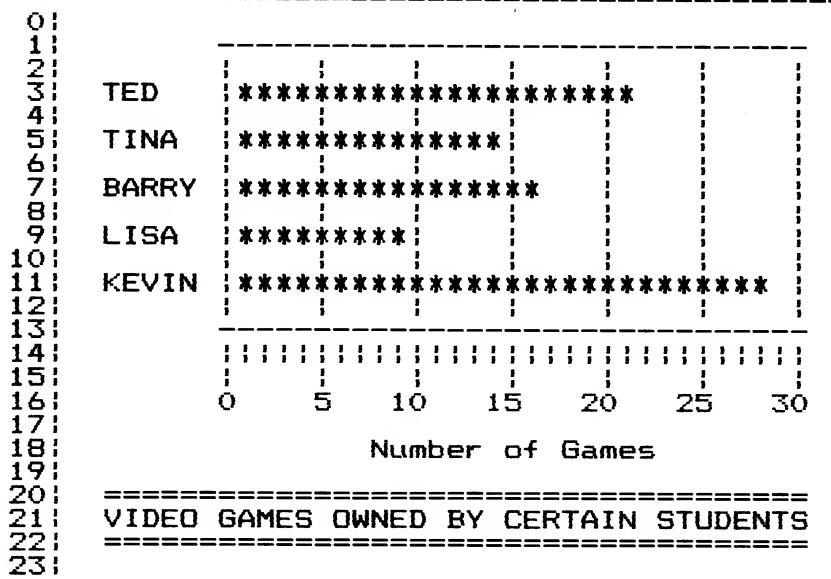
FIGURE 18-1   Design of Screen Display

Here's the beginning part of the program. We've put a large number of REMark statements in it for your help. You may leave them out at this point, but you will want to add them before you save the program on a disk.

```
100 REM ********************************
101 REM **                            **
102 REM **          BARGRAPH          **
103 REM ** Review Demonstration #1    **
104 REM **     J. Reisinger 1983      **
105 REM **                            **
106 REM ********************************
110 GRAPHICS 0
114 REM ** Get rid of cursor          **
115 POKE 752,1
120 POSITION 2,0
125 PRINT #6;"  "
127 REM ** N$      - individual name  **
128 REM ** NAME$ - 5 names            **
129 REM ** PITCH - sound value        **
130 DIM N$(5),NAME$(25),PITCH(5)
133 REM ** Dark blue text on light    **
134 REM ** blue background            **
135 SETCOLOR 2,7,8
140 SETCOLOR 4,7,8
145 SETCOLOR 1,7,2
```

```
146 REM ** Name the line numbers    **
147 REM ** where the subroutines    **
148 REM ** start                     **
150 PITCHVAL=200
151 NAMEREAD=300
152 DRAWGRID=400
153 HORSCALE=500
154 NAMEBARS=600
155 TITLEPRT=700
159 REM **** Begin main program *****
160 GOSUB PITCHVAL
165 GOSUB NAMEREAD
170 GOSUB DRAWGRID
175 GOSUB HORSCALE
180 GOSUB NAMEBARS
185 GOSUB TITLEPRT
190 GOTO 190
191 REM ****** end of program *******
```

Note lines 150 to 155. We chose an eight-letter variable name for each subroutine and assigned the beginning line number of the subroutine to that variable name. This gives us a way to refer to subroutines by name rather than number.

Now add the first three subroutines and the program DATA (line 799 to 810).

```
195 REM ******************************
196 REM **                          **
197 REM ** PITCHVAL - subroutine    **
198 REM **                          **
199 REM ******************************
200 FOR P=1 TO 5
205 READ TONE
210 PITCH(P)=TONE
215 NEXT P
220 RETURN
295 REM ******************************
296 REM **                          **
297 REM ** READNAME - subroutine    **
298 REM **                          **
299 REM ******************************
300 FOR N=1 TO 5
305 READ N$:L=LEN(N$)
310 IF L=5 THEN 330
315 FOR C=L+1 TO 5
320 N$(C,C)=" "
325 NEXT C
330 NAME$(5*N-4,5*N)=N$
335 NEXT N
```

```
340 RETURN
392 REM ****************************
393 REM **                        **
394 REM ** DRAWGRID - subroutine   **
395 REM **                        **
396 REM ****************************
397 REM
398 REM ** Draw top and bottom    **
399 REM ** lines of grid          **
400 FOR X=8 TO 37
405 POSITION X,1:PRINT #6;CHR$(14)
410 POSITION X,13:PRINT #6;CHR$(13)
415 NEXT X
418 REM ** Draw vertical lines of **
419 REM ** grid                   **
420 FOR Y=2 TO 12
425 FOR X=8 TO 38 STEP 5
430 POSITION X,Y
435 PRINT #6;CHR$(22)
440 NEXT X
445 NEXT Y
450 RETURN
799 REM ** Sound DATA             **
800 DATA 121,96,81,60,47
804 REM ** Student names          **
805 DATA TED,TINA,BARRY,LISA,KEVIN
809 REM ** Number of games/student **
810 DATA 21,14,16,9,27
```

You can run this much of the program by adding this line:

```
171 GOTO 171
```

Be sure to delete it again before you continue.

Let's consider the subroutine DRAWGRID. It draws the background grid for the graph: a top and bottom border and vertical lines representing 5 units. If we refer to Figure 18-1, we see that the horizontal borders should extend from column 8 to column 38. Now look at line 400:

```
400 FOR X=8 TO 37
```

Why 37 instead of 38? Well, the character we're using for the vertical lines, CTRL-V, prints a vertical line in the extreme left position of the pixel. Therefore, we need to go only to column 37. Change the 37 in line 400 to 38 to see the difference. We print the top border and bottom border simultaneously at lines 405 and 410. The characters printed are CTRL-N and CTRL-M, respectively.

To draw the vertical lines, we use nested FOR...NEXT loops (lines 420 to 445). We can see from our picture that the vertical lines extend from row 2 to row 12 (the Y loop), and occupy columns 8, 13, 18, 23, 28, 33, and 38; that is, every five columns, starting at column 8.

Now add the subroutine HORSCALE to add the horizontal scale to the graph:

```
493 REM **************************
494 REM **                      **
495 REM ** HORSCALE - subroutine **
496 REM **                      **
497 REM **************************
498 REM
499 REM ** Print scale          **
500 FOR X=0 TO 30
505 POSITION X+8,14
510 PRINT #6;CHR$(22)
515 IF INT(X/5)<>X/5 THEN 545
520 POSITION X+8,15
525 PRINT #6;CHR$(22)
529 REM ** Print numbers on scale **
530 X1=X-1:IF X<10 THEN X1=X1+1
535 POSITION X1+8,16
540 PRINT #6;X
545 NEXT X
549 REM ** Print horizontal label **
550 POSITION 16,18
555 PRINT #6;"Number of Games"
560 RETURN
```

The subroutine HORSCALE draws the scale at the bottom of the graph and prints the numbers 0, 5, 10, 15, 20, 25, and 30 in the appropriate places. We chose to print all three rows of the graph (14, 15, and 16) in one loop. The X represents the number of the column on the screen; thus at line 505, the first coordinate in the POSITION statement is X+8 because the first vertical line is in column 8. At line 515, we check to see if we need to print something in rows 15 and 16. If not, we go to the next column. At line 530, we adjust the X-coordinate to print either a 1-digit number (0, 5) or a 2-digit number (10, 15, 20, 25, 30). To run this part of the program, add this line:

```
176 GOTO 176
```

The subroutine NAMEBARS is the real heart of the program. It prints the students' names and then draws a bar of the appropriate length to represent the number of video games owned.

```
595 REM **********************************
596 REM **                            **
597 REM ** NAMEBARS - subroutine       **
598 REM **                            **
599 REM **********************************
600 FOR N=1 TO 5
604 REM ** Print name                 **
605 N$=NAME$(5*N-4,5*N)
610 Y=2*N+1
615 POSITION 2,Y
620 PRINT #6;N$
625 READ NUMBER
629 REM ** Print bar for student N **
630 FOR COLUMN=1 TO NUMBER
635 POSITION 7+COLUMN,Y
640 PRINT #6;CHR$(160)
644 REM ** Add sound to printing     **
645 SOUND 0,PITCH(N),10,10
650 FOR W=1 TO 8:NEXT W
655 SOUND 0,0,0,0
660 FOR W=1 TO 2:NEXT W
665 NEXT COLUMN
670 NEXT N
675 RETURN
```

You'll recall that we stored all of the students' names in the string NAME$. At lines 605 to 620, we retrieve the name and print it in the correct row. The number of the row is one more than twice the number of the student (line 610).

Lines 630 to 665 print the bar of the appropriate length. We've included some sound for interest. We stored the pitch values for the sound back in the PITCHVAL subroutine at lines 195 to 220.

Here's the last subroutine, the title for the graph:

```
695 REM **********************************
696 REM **                            **
697 REM ** TITLEPRT - subroutine      **
698 REM **                            **
699 REM **********************************
700 FOR X=2 TO 38
705 POSITION X,20
710 PRINT #6;CHR$(18)
715 POSITION X,22
720 PRINT #6;CHR$(18)
725 NEXT X
730 POSITION 2,21
735 PRINT #6;"VIDEO GAMES OWNED BY CERTAIN STUDENTS"
740 RETURN
```

Now that it's all together, we certainly hope that your graph looks like the picture on page 246.

In Chapter 17 we considered a pair of subroutines associated with drawing circles. We'd like to explain a bit of the math behind these subroutines so that we can discuss a program to draw discs, or "filled" circles.

First, imagine a unit circle ("unit" means that the radius is equal to 1) placed in the coordinate plane with its center at the origin (**0,0**). Here's a sketch (Figure 18-2).
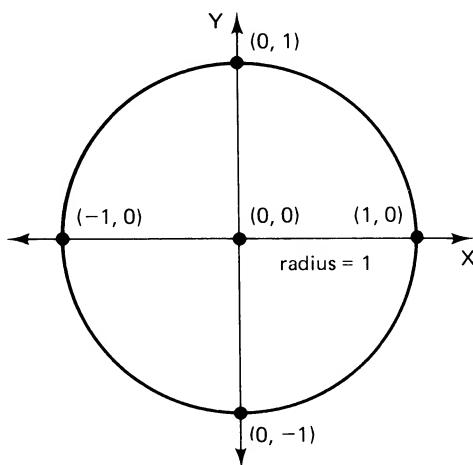
FIGURE 18-2   Unit Circle

Now, suppose that a point P with coordinates (x,y) moves counterclockwise around the circle, starting at point A (**1,0**). A radius is drawn from point P to the center of the circle. This radius makes an angle A99 with the positive X-axis. As point P moves around the circle, angle A99 increases from 0 degrees to 360 degrees. At point B, the angle is 90 degrees; at point C, it is 180 degrees; and at point D, it is 270 degrees. Figure 18-3 on page 252 shows this.

The question now is, "What are the coordinates (x,y) of point P?" The answer is simple:

$$x = cosine(A99)$$
$$y = sine(A99)$$

This is the mathematical definition of the cosine and sine functions (abbreviated cos and sin). These functions are part of ATARI BASIC; that is, the computer can calculate the value of COS(A99) and SIN(A99) for all values of the angle A99.
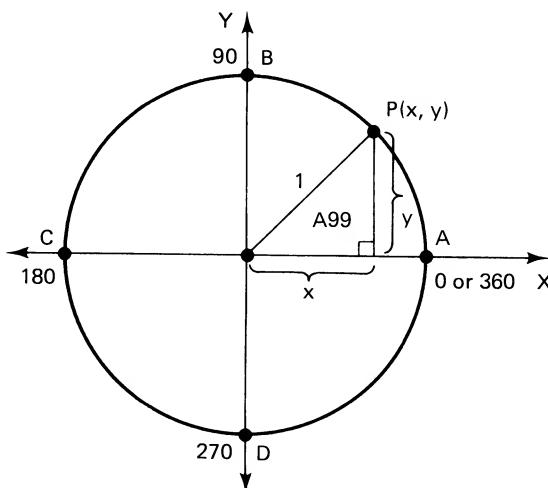
FIGURE 18-3 Coordinates of Point on Unit Circle

Now we can understand the subroutine CIRCLE COORDI-NATES in Chapter 17. We calculate the cosine and sine of each whole number angle from 0 to 360 degrees and store the result in a subscripted variable. The subscript is the same as the number of degrees.

Remember that the coordinates we've stored are for points on a *unit circle*. If the circle has a radius of 10, then the coordinates are 10 times as big; if the circle has a radius of 50, then the coordinates are 50 times as big, etc. (If you have studied geometry, you can easily see that this follows from the properties of similar triangles.)

To draw a circle after we've stored the coordinates, we repeat these steps for each value of the angle from 0 degrees to 360 degrees:

1. **Retrieve the coordinates from the subscripted variables**

2. **Multiply the coordinates by the radius of the circle**

3. **Round the coordinates to the nearest integer**

4. **ADD the x-coordinate to the x-coordinate of the center of the circle**

5. **SUBTRACT the y-coordinate from the y-coordinate of the center**

6. **Plot the resulting point**

Only one problem arises: The result is slightly egg-shaped! This is because the screen pixels are not quite square—they are a bit taller than they are wide. We correct this by multiplying the x-coordinate by 1.2 before we round it at step 3.

Have you forgotten our original problem? We want to draw a disc instead of just the circumference of a circle. Run the follow-

ing program. Don't type NEW when you're finished, because
we're going to add to it.

```
100 REM *****************************
101 REM **                        **
102 REM **       FILLCIRC         **
103 REM ** Review Demonstration #2 **
104 REM **   J. Reisinger 1983     **
105 REM **                        **
106 REM *****************************
130 C=1
139 REM ** Name subroutines        **
140 FINDCOORD=800
150 CENTERRAD=300
160 FILLCIRCL=400
170 RINGCIRCL=600
195 REM *****************************
196 REM **                        **
197 REM **      MAIN PROGRAM       **
198 REM **                        **
200 GOSUB FINDCOORD
210 GRAPHICS 7+16
220 COLOR 1
230 FOR A=0 TO 360
240 X=INT(1.2*35*XCIR(A)+0.5)
250 Y=INT(35*YCIR(A)+0.5)
260 PLOT 80,40:DRAWTO 80+X,40-Y
270 NEXT A
280 GOTO 280
795 REM *****************************
796 REM **                        **
797 REM ** FINDCOORD - Subroutine  **
798 REM **                        **
799 REM *****************************
800 DIM XCIR(360),YCIR(360)
810 DEG
820 GRAPHICS 2
830 SETCOLOR 2,0,0
840 POSITION 5,5
850 PRINT #6;"COUNT DOWN"
860 FOR W=1 TO 500:NEXT W
870 POSITION 5,5
880 PRINT #6;"            "
890 FOR A99=0 TO 360
900 X99=COS(A99)
910 Y99=SIN(A99)
920 XCIR(A99)=X99
930 YCIR(A99)=Y99
940 POSITION 8,5
950 PRINT #6;360-A99;"   "
960 NEXT A99
970 RETURN
```

We got a disc, all right, but it has holes in it because not all the points inside the circle lie on the spokes that we drew. Let's consider another possibility. The fill command that we used in Chapter 12 worked well for areas bounded by straight lines, but it is quite tedious to use with curved lines. We'll settle for the "windowshade" method. By this we mean that we'll draw the disc as a series of horizontal stripes from top to bottom.

Add the following lines to your program now, and then we'll ask you questions about how it works in the exercises for this chapter. You can think of this program and the exercises as a type of final exam.

```
210 GRAPHICS 7+16
220 SETCOLOR 0,4,4
230 SETCOLOR 1,0,10
240 GOSUB CENTERRAD
250 GOSUB FILLCIRCL
260 GOSUB RINGCIRCL
270 C=C+1
280 IF C=4 THEN C=0
285 GOTO 240
290 REM **                           **
291 REM **     END OF MAIN PROGRAM    **
292 REM **                           **
293 REM ***************************
294 REM
295 REM ***************************
296 REM **                           **
297 REM ** CENTERRAD - subroutine  **
298 REM **                           **
299 REM ***************************
300 H=INT(160*RND(0))
310 K=INT(96*RND(0))
320 R=5+5*INT(5*RND(0)+1)
330 COLOR C
340 RETURN
395 REM ***************************
396 REM **                           **
397 REM ** FILLCIRCL - subroutine  **
398 REM **                           **
399 REM ***************************
400 FOR A=90 TO 270
410 X1=H+INT(1.2*R*XCIR(A)+0.5)
420 Y1=K-INT(R*YCIR(A)+0.5)
430 IF Y1<0 THEN 540
440 IF Y1>95 THEN 550
450 IF A>180 THEN 490
460 X2=H+INT(1.2*R*XCIR(180-A)+0.5)
470 Y2=K-INT(R*YCIR(180-A)+0.5)
480 GOTO 510
```

```
490 X2=H+INT(1.2*R*XCIR(540-A)+0.5)
500 Y2=K-INT(R*YCIR(540-A)+0.5)
510 IF X1<0 THEN X1=0
520 IF X2>159 THEN X2=159
530 PLOT X1,Y1:DRAWTO X2,Y2
540 NEXT A
550 RETURN
595 REM ********************************
596 REM **                          **
597 REM ** RINGCIRCL - subroutine   **
598 REM **                          **
599 REM ********************************
600 IF C=0 THEN C1=2:GOTO 620
610 C1=0
620 COLOR C1
630 FOR A1=0 TO 360
640 X=H+INT(1.2*R*XCIR(A1)+0.5)
650 Y=K-INT(R*YCIR(A1)+0.5)
660 IF X<0 OR X>159 THEN 690
670 IF Y<0 OR Y>95 THEN 690
680 PLOT X,Y
690 NEXT A1
700 RETURN
```

```
100 REM - Chapter 3, No. 3
110 GRAPHICS 2:SETCOLOR 2,0,0
120 FOR K=4 TO 6 STEP 2
130 FOR M=8 TO 10
140 READ FINAL
150 POSITION M,K
160 PRINT #6;CHR$(FINAL+68)
170 NEXT M
180 NEXT K
190 GOTO 190
200 DATA 16,4,1,1,10,0
```

The Final Program

EXERCISES

1. Modify the bar graph program so that the bars are vertical instead of horizontal. We have kept the title of the graph in the same position, so the subroutine TITLEPRT won't need to be changed at all.

The key idea to keep in mind for changing the subroutines DRAW-GRID, HORSCALE, and NAMEBARS is to exchange the order of the row and column loops.

Here is the sample display. Some hints for modifying the program follow.

```
                    (read column numbers down)
          01234567891111111111222222222233333333333
                    0123456789012345678901234567889
          ------------------------------------------------
     0 |                                                !
     1 |                                                !
     2 | N 30-|------------------------------------|  !
     3 | U   -|                                    |  !
     4 | M   -|                            ****    |  !
     5 | B   -|                            ****    |  !
     6 | E   -|                            ****    |  !
     7 | R 20-|-****-----------------------****--|  !
     8 |     -| ****                       ****    |  !
     9 | o   -| ****           ****        ****    |  !
    10 | f   -| ****  ****     ****        ****    |  !
    11 |     -| ****  ****     ****        ****    |  !
    12 | G 10-|-****--****--****--------****--|  !
    13 | A   -| ****  ****  ****  ****     ****    |  !
    14 | M   -| ****  ****  ****  ****     ****    |  !
    15 | E   -| ****  ****  ****  ****     ****    |  !
    16 | S   -| ****  ****  ****  ****     ****    |  !
    17 |    -0-|-****--****--****--****--****--|  !
    18 |        TED   TINA  BARRY LISA  KEVIN     !
    19 |                                            !
    20 | ===================================== !
    21 | VIDEO GAMES OWNED BY CERTAIN STUDENTS !
    22 | ===================================== !
    23 |                                            !
          ------------------------------------------------
```

Hints for modifying the subroutine DRAWGRID:

a. The top and bottom lines of the grid start at column 3 and end at column 38. They are in rows 2 and 17. (See lines 398 to 415.)

b. There are just two horizontal lines inside the grid, at rows 7 and 12. These lines extend from column 7 to column 37. (See lines 418 to 445.)

Hints for modifying the subroutine HORSCALE:

a. Because this scale is vertical, we should change the name of the subroutine to VERSCALE.

b. We need dashes in column 5 from rows 2 to 17. (See lines 499 to 525.)

c. We need to print the numbers 30, 20, 10, and 0 at rows 2, 7, 12, and 17. (See lines 529 to 545.) A formula for this number in terms of the ROW (ROW varies from 2 to 17) is NUM=34-2*ROW. Try it!

d. We will need a loop to print the vertical label NUMBER of GAMES in column 1 starting at row 2. Clever (?) suggestion: If we stored NUMBER of GAMES in a string DIMensioned for 16 characters (the last character being a space), then we could print the label with the same loop that we used to print the dashes in column 5.

Hints for modifying the subroutine NAMEBARS:

a. The first letters of the names are printed in columns 8, 14, 20, 26, and 32; that is, every 6 columns. These numbers also indicate the beginning columns for each of the bars, which are four columns wide. If you start a loop with the statements

```
600 FOR X=8 TO 32 STEP 6
602 N=(X-2)/6
605 N$=NAME$(5*N-4,5*N)
```

then you can print the names with the statements

```
615 POSITION X,18
620 PRINT #6;N$
```

b. After we read the number of games for a student, we need to calculate the number of the top row of that student's bar. Keep in mind that each vertical line of our graph represents two games, so we'll have to do some rounding. You can see from our sample graph that if the number of games was odd, then we rounded to the next lower even number. Thus, 21, 9, and 27 were rounded to 20, 8, and 26, respectively. A formula for this is

```
NUMBER=2*INT(NUMBER/2)
```

Now you can calculate the number of the top row, and complete the subroutine.

The following questions deal with the final version of the FILLCIRC program.

2. What is the function of line 830?

_____

_____

3. What is the function of line 880?

_____

_____

4. The subroutine CENTERRAD does four things. What are they?

_____

_____

_____

_____

5. At line 950, why do we add " " at the end of the line?

_____

6. Why can't the following four lines

```
900  X99=COS(A99)
910  Y99=SIN(A99)
920  XCIR(A99)=X99
930  YCIR(A99)=Y99
```

be replaced with just these two lines?

```
900  XCIR(A99)=COS(A99)
910  YCIR(A99)=SIN(A99)
```

_____

7. What is the function of lines 660 and 670 in the subroutine RINGCIRCL?

_____

_____

8. Explain what is represented by the variables X1, Y1, X2, and Y2 in the subroutine FILLCIRCL.

X1 = _____

Y1 = _____

X2 = _____

Y2 = _____

9. Suppose that the angle represented by A in the subroutine FILL-CIRCL is equal to 115 degrees. The coordinates of the *left* endpoint of the horizontal stripe associated with this angle are

   a. (X1,Y1)   **b.** (X2,Y2)   **c.** (H,K)   **d.** (H+X1,K-Y1)

_____

10. In Problem 9, what is the angle associated with the *right* endpoint of the stripe determined by an angle of 115 degrees?

_____

11. Explain the function of the subroutine RINGCIRCL.

_____

_____

12. Modify the subroutine FILLCIRCL so that the disks are drawn with vertical stripes instead of horizontal ones. Hint: Use the following FOR...TO statement at line 400:

```
400 FOR A=0 TO 180
```
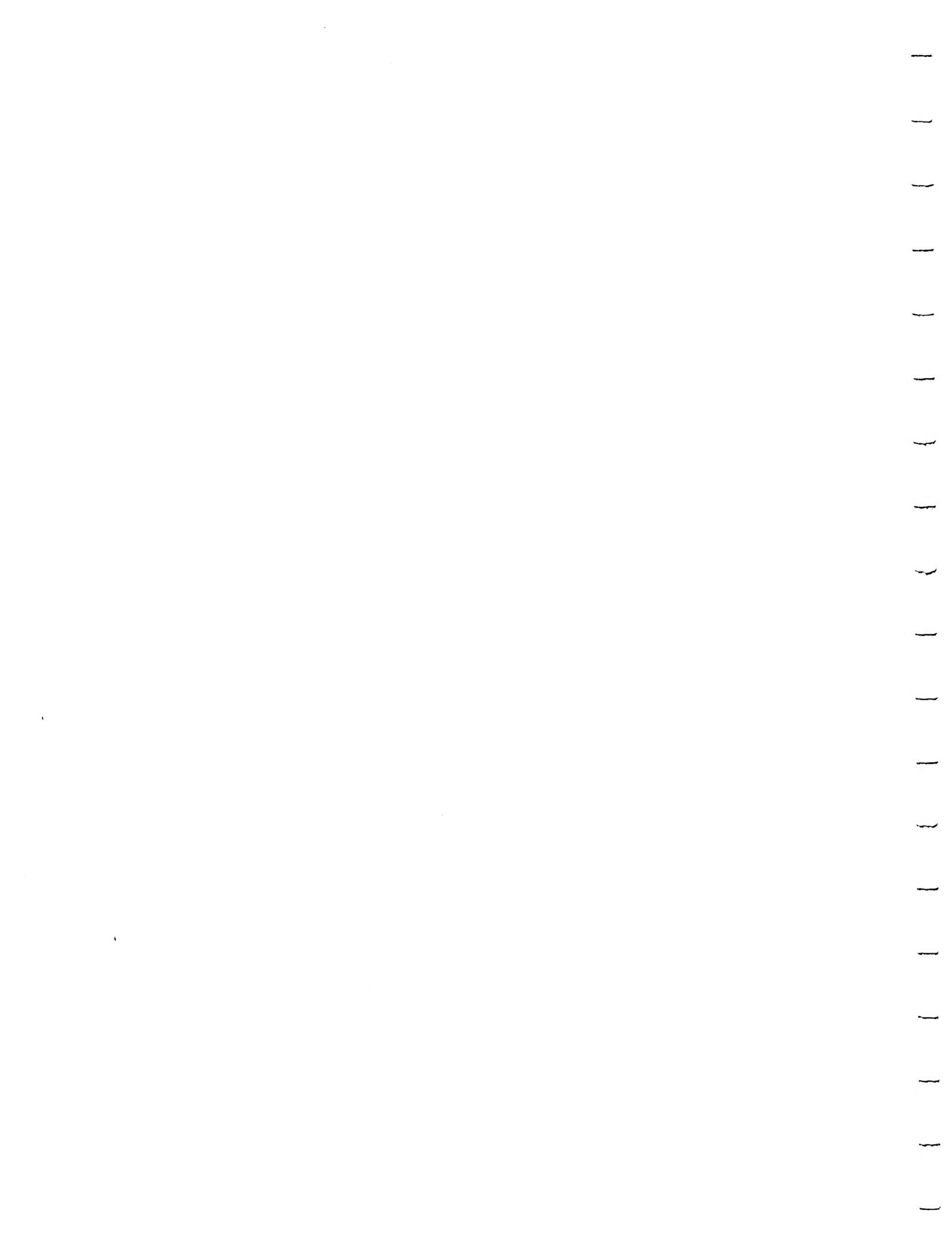
Error Messages

ERROR CODE NO.	ERROR CODE MESSAGE
2	**Memory Insufficient.** Not enough RAM to store the statement of the new variable name or to DIM a new string variable. Possible remedy: Delete REM statements, combine other statements, reduce amount of text to be printed.
3	**Value Error.** A value expected to be a positive integer is negative; a value expected to be within a specific range is not.
4	**Too Many Variables.** A maximum of 128 different variable names is allowed.
5	**String Length Error.** Attempted to store beyond the DIMensioned string length.
6	**Out of Data Error.** READ statement requires more data items than supplied by DATA statement(s).
7	**Number Greater than 32767.** Value is not a positive integer or is greater than 32767.
8	**Input Statement Error.** Attempted to INPUT a non-numeric value into a numeric variable.
9	**Array or String DIM Error.** DIM size is greater than 32767 or an array/matrix reference is out of the range of the DIMensioned size, or the array/matrix or string has already been DIMensioned, or a reference has been made to an undimensioned array or string.

10	**Argument Stack Overflow.**
	There are too many GOSUBs or too large an expression.
11	**Floating Point Overflow/Underflow Error.**
	Attempted to divide by zero or refer to a number larger than 10 to the 98th power or smaller than 10 to the -99th power.
12	**Line Not Found.**
	A GOSUB, GOTO, or THEN statement referred to a nonexistent line number.
13	**No Matching FOR Statement.**
	A NEXT was encountered without a previous FOR, or nested FOR...NEXT statements do not match properly. (Error is reported at the NEXT statement, not at FOR.)
14	**Line Too Long Error.**
	The statement is too complex or too long for BASIC to handle.
15	**GOSUB or FOR Line Deleted.**
	A NEXT or RETURN statement was encountered and the corresponding FOR or GOSUB has been deleted since the last RUN.
16	**RETURN Error.**
	A RETURN was encountered without a matching GOSUB.
17	**Garbage Error.**
	Execution of "garbage" (bad RAM bits) was attempted. This error code may indicate a hardware problem but may also be the result of faulty use of POKE. Try typing NEW or powering down, then reenter the program without any POKE commands.
18	**Invalid String Character.**
	String does not start with a valid character, or string in VAL statement is not a numeric string.
Note:	**The following are INPUT/OUTPUT errors that result during the use of disk drives, printers, or other accessory devices. Further information is provided with the auxiliary hardware.**
19	**LOAD Program Too Long.**
	Insufficient memory remains to complete LOAD.
20	**Device Number Error.**
	The device number is larger than 7 or equal to 0.
21	**LOAD File Error.**
	Attempted to LOAD a non-LOAD file.
128	**BREAK Abort.**
	User hit the BREAK key during an INPUT/OUTPUT operation.
129	**IOCB Error.**
	Input/Output Control Block (IOCB) already open.

130	**Nonexistent Device Error.** Referred to a nonexistent device.
131	**IOCB Write Only.** READ command to a write-only device (printer).
132	**Invalid Command.** The command is invalid for this device.
133	**Device or File not Open.** No OPEN specified for this device.
134	**Bad IOCB Number.** Illegal device number.
135	**IOCB Read Only Error.** WRITE command to a read-only device.
136	**EOF.** End of File (EOF) has been reached. (NOTE: This message may occur when using cassette files.)
137	**Truncated Record.** Attempt to read a record longer than 256 characters.
138	**Device Timeout.** Device doesn't respond. (Is it plugged in and turned on?)
139	**Device NAK.** Garbage at serial port or bad disk drive.
140	**Serial bus** input framing error.
141	**Cursor out of range** for this particular (graphics) mode.
142	**Serial bus data frame overrun.**
143	**Serial bus data frame checksum error.**
144	**Device Done Error.** (Invalid "done" byte): Attempt to write on a write-protected diskette.
145	**Read after write compare error.** (Disk handler) or bad screen mode handler.
146	**Function not implemented** in handler.
147	**Insufficient RAM** for operating selected graphics mode.
160	**Drive number error.**
161	**Too many OPEN files.** (No sector buffer available.)
162	**Diskette full.** (No free sectors - ca. 707 sectors maximum per diskette.)
163	**Unrecoverable system data INPUT/OUTPUT error.**
164	**File number mismatch.** Links on diskette are messed up.

165	File name error.
166	POINT data length error.
167	File locked.
168	Command invalid. (Special operation code.)
169	Directory full. (64 files maximum per diskette.)
170	File not found.
171	POINT invalid.

Pitch Values for Sound Statements

PITCH NUMBER	NOTE	PITCH NUMBER	NOTE
14	C (very high)		
15	B	64	B
16	A# or B*b*	68	A# or B*b*
17	A	72	A
18	G# or A*b*	76	G# or A*b*
19	G	81	G
20	F# or G*b*	85	F# or G*b*
21	F	91	F
23	E	96	E
25	D# or E*b*	102	D# or E*b*
26	D	108	D
28	C# or D*b*	114	C# or D*b*
29	C (high)	121	C (middle)
31	B	129	B
33	A# or B*b*	137	A# or B*b*
35	A	145	A
37	G# or A*b*	153	G# or A*b*
40	G	163	G
42	F# or G*b*	173	F# or G*b*
44	F	183	F
47	E	193	E
50	D# or E*b*	204	D# or E*b*
53	D	217	D
57	C# or D*b*	229	C# or D*b*
60	C	243	C (low)

```
_C__do ___29_
 B   ti    31
_A__la ____35_
 G   sol   40
_F___fa ___44_____
 E   mi    47
_D__re ___53_____          ← PITCH VALUES ON THE C SCALE
 C   do    60
_B__ti ___64_____
 A   la    72
_G__sol__ 81_____
 F   fa    91
_E__mi___96_____
 D   re    108
_C__do__ 121_(middle C)
```

Appendix 3

Introduction to DOS II

DOS II (Disk Operating System) is a two-part program that is contained on the ATARI Master Diskette II. This program allows the ATARI computer to communicate with the ATARI Disk Drive so that you can:

1. **Store programs on diskettes.**
2. **Retrieve programs from diskettes.**
3. **Create data and add to data files on diskettes.**
4. **Make copies of entire diskettes or of individual files on diskettes.**
5. **Delete old files from a diskette.**
6. **Examine the directory of files on a diskette.**
7. **Format a new diskette so that it can be used with the ATARI.**

Items 1) and 2) are discussed in Chapter 1; all of the others use the DOS II Menu. To call up the Menu on the screen, first load DOS as outlined in Chapter 1. When the word READY appears on the screen, type the word DOS and then press RETURN. After a few seconds, the display in Figure Appendix 3-1 will appear on the screen. The DOS Menu gives us a list of disk applications. To choose an application, we simply type the letter of the application and press the RETURN key. We'll describe the selections that you will probably want to use as a beginner; all of the selections are described in detail in the DISK OPERATING SYSTEM II REFERENCE MANUAL which is provided with the ATARI disk drive. We have indicated the responses you should make to the computer prompts by means of an arrow (→).

DOS Menu

```
DISK OPERATING SYSTEM II VERSION 2.0S
COPYRIGHT 1980 ATARI

A. DISK DIRECTORY       I. FORMAT DISK
B. RUN CARTRIDGE        J. DUPLICATE DISK
C. COPY FILE            K. BINARY SAVE
D. DELETE FILE(S)       L. BINARY LOAD
E. RENAME FILE          M. RUN AT ADDRESS
F. LOCK FILE            N. CREATE MEM. SAV
G. UNLOCK FILE          O. DUPLICATE FILE
H. WRITE DOS FILES



SELECT ITEM OR RETURN FOR MENU
```
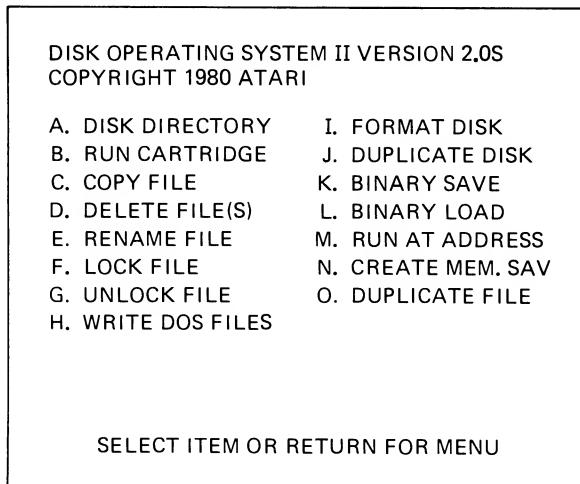
A. Disk Directory

> **SELECT ITEM OR RETURN FOR MENU**
> ➤ **A (RETURN)**
> **DIRECTORY--SEARCH SPEC,LIST FILE?**
> ➤ **(RETURN)**
>
> **DOS   SYS 039**
> **DUP   SYS 039**
> **626 FREE SECTORS**
>
> **SELECT ITEM OR RETURN FOR MENU**

This example shows the results if you are looking at the directory of the Master Diskette; it contains only the two parts of DOS II. The number after the name of each file shows how many sectors the file occupies—one sector represents approximately 125 characters of information. The number of unused (free) sectors left on the diskette is printed at the end of the directory.

If you would rather have a hard copy of the disk directory (a handy thing to have as the number of your diskettes increases), then turn your printer on and give the following responses:

> **SELECT ITEM OR RETURN FOR MENU**
> ➤ **A (RETURN)**
> **DIRECTORY--SEARCH SPEC,LIST FILE?**
> ➤ **D:,P: (RETURN)**
> **(the directory is printed on the printer)**
> **SELECT ITEM OR RETURN FOR MENU**

I. Format Disk

Each new diskette must have a particular pattern of magnetism on it so that it can be used by the ATARI computer. This pattern is not the same for all micro computers. The FORMAT DISK application will create the proper magnetic pattern for the ATARI on a new disk, or ERASE all of the files on an old disk so that you may use it again for some other programs. It is a good practice to check the directory of any used disk before you format it; otherwise, you may lose valuable programs.

> **SELECT ITEM OR RETURN FOR MENU**
> ➤ **I (RETURN)**
>   **WHICH DRIVE TO FORMAT?**
> ➤ **1 (RETURN)**
>   **TYPE "Y" TO FORMAT DISK 1**
> ➤ **Y (RETURN)**
>   **(disk drive whirs for about 40 seconds)**
>   **SELECT ITEM OR RETURN FOR MENU**
> ➤ **A (RETURN)   (RETURN)**
>   **707 FREE SECTORS**

The last line indicates that the disk has been formatted properly. (707 is the magic number to watch for.) It is always a good idea to use the directory to verify the formatting.

H. Write DOS Files

If you would like to put the DOS program on your own disk so that you will not always have to hunt for the Master Diskette in order to load DOS, then do the following:

> **SELECT ITEM OR RETURN FOR MENU**
> ➤ **H (RETURN)**
>   **DRIVE TO WRITE DOS FILES TO?**
> ➤ **1 (RETURN)**
>   **TYPE "Y" TO WRITE DOS TO DRIVE 1**
> ➤ **Y (RETURN)**
>   **(disk drive whirs for about 35 seconds)**
>   **SELECT ITEM OR RETURN FOR MENU**
> ➤ **A (RETURN) (RETURN)**
>   **check to see if the directory contains**
>   **DOS   SYS 039**
>   **DUP   SYS 039**

D. Delete File(s)

Perhaps you have a program that you no longer want stored on a disk. Selection D. allows you to erase the program (file) so that you can use the space for something else.

SELECT ITEM OR RETURN FOR MENU
➤ **D (RETURN)**
DELETE FILE SPEC
➤ **PROGRAM1 (RETURN)**
　　　　　　　(type the name of the program to be deleted)
TYPE "Y" TO DELETE...
D1:PROGRAM1.　　?
➤ **Y (RETURN)**
SELECT ITEM OR RETURN FOR MENU
➤ **A (RETURN) (RETURN)**
(check to see if the program is deleted from the directory.)

B. Run Cartridge

If you have been using the DOS II Menu and you want to return to BASIC programming, then do the following:

SELECT ITEM OR RETURN FOR MENU
➤ **B (RETURN)**
the DOS II Menu is replaced with

READY

Glossary

algorithm
A set of steps or a procedure to follow to perform a task (usually a repetitive task). In programs, algorithms are usually represented by loops.

argument
The constant or variable operated on by a function.

arithmetic operator
One of the symbols +, *, -, /, ∧ used to indicate a calculation to be performed on two numbers.

array
A list of numerical values stored in a series of memory locations. The items are usually referred to by a subscripted variable.

BASIC
Beginners All-purpose Symbolic Instruction Code. A programming language for computers.

BASIC language cartridge
A small piece of hardware that contains the BASIC language in electronic form.

binary number
A number expressed in base 2 notation instead of base 10 notation. Binary numbers are written using only the digits 0 and 1. For example, the base 10 number 25 is written 11001 as a binary number.

body of a loop
The set of statements in a FOR...NEXT loop between the FOR and the NEXT statements.

branch
A change in the normal sequence of execution of program statements, so that the program either skips over some lines or returns to a previous line. Also referred to as transfer of control.

byte
The unit of measure for computer memory, usually one character
(letter, numeral, punctuation mark, etc.) of information.

central processing unit (CPU)
The part of the computer that controls the memory and the
peripheral devices. In the ATARI, part of the 6502 microproces-
sor "chip."

character keys
The keys on the computer keyboard that correspond to normal
typewriter keys.

color register
A memory location that holds color information for the graphics
modes.

command
An instruction to the computer that is executed immediately, for
example, RUN.

compact spacing
A printing option in which items are printed with no spaces
separating them.

concatenate
To attach one string on to the end of another string, producing a
new string.

condition
A statement involving a variable or variable expression that is
tested for purposes of branching. For example, X=3 is the condi-
tion in the statement IF X=3 THEN 220.

conditional branch
A branch in which there is more than one option: control may
pass to one of several different lines, depending on whether the
condition is true or false.

control characters
The cursor control arrows and the clear screen arrow.

control graphics characters
A set of 29 "picture" characters that can be displayed by holding down the CONTROL key while pressing one of the alphabet keys or comma (,), period (.), or semi-colon (;).

counter
A numerical variable in a program that is incremented by one each time a particular event occurs, so that the occurrences of the event are counted.

cursor
A square displayed on the TV or monitor screen that shows where the next typed character will be displayed.

data statement
A statement in a program that holds information (data).

decision
A program statement which tests the value of a variable or variable expression to see if control should be transferred.

decrement
(noun): the amount by which a variable is decreased in a loop.
(verb): to decrease the value of a variable in a loop.

default colors
The "built-in" colors for graphics modes that are used unless the user gives the computer instructions to use other colors.

default value
A "built-in" numerical value that the computer uses unless it is changed by the user.

deferred mode
An operating mode in which line numbers are used with BASIC statements. The statements are not executed until the RUN command is given.

device number
A number used to identify a peripheral device, such as a disk drive. #6, for example, refers to the screen when we use the text modes (GRAPHICS 1 or GRAPHICS 2).

direct mode
An operating mode in which no line number is used for an instruction, and in which the instruction is executed as soon as the RETURN key is pressed.

disk
A record/playback medium like tape, but made in the shape of a flat disk that is placed inside a stiff envelope for protection. Disks for the ATARI are 5 1/4 inches in diameter.

disk drive
A peripheral hardware device for storing information on disks and retrieving information from them.

diskette
See disk.

distortion
1) A non-musical tone, or
2) the 3rd number in a SOUND command, that determines whether the sound will be a musical tone or will be distorted.

documentation
Any written material that explains or describes a program.

DOS
Abbreviation for "disk operating system". A program that allows the computer to communicate with the disk drive. Pronounced "doss".

double-subscripted variable
A variable with two subscripts; one indicating the row, and the other indicating the column of the element in a matrix.

drive code switch
A hardware switch on a disk drive that allows the user to designate the drive as number 1, 2, 3 or 4.

duration
The length of time that a note in a song is played.

endless loop
A loop which will be repeated until the user intervenes by pressing the BREAK key or by turning off the computer.

error message
A response by the computer, printed on the screen, that indicates that a particular type of mistake has been made. (See Appendix 1.)

execute mode
Also called RUN mode. After the RUN command is given, each program line is processed and executed.

exponentiation
The process of raising a number to a power.

extender
A maximum of 3 letters that may be added to a file name for identification purposes.

factorial
For any positive integer, the product of the integer and all of the integers less than it. The factorial of a number N is written N!. For example, 4! = 4 × 3 × 2 × 1 = 24.

file
A program or list of data stored on a disk.

filename
The name (maximum 8 letters plus 3 letters for the extender) of a file on a disk.

filespec
Short for file specification.

file specification
The complete file designation, including a reference to the disk drive and the filename.

flag

A specific value of a variable that is used in a decision to test whether or not branching should occur. Flags are often used at the end of a series of DATA statements to indicate that all of the DATA has been used.

flexible disk

See disk.

flowchart

A diagram or chart which shows the logical structure of a program.

format

1) (verb): to prepare a new disk so that it can be used with the disk drive. FORMAT is option I on the DOS menu. (See Appendix 4.)

2) (noun): the size of the screen in a particular graphics or text mode.

full screen

A screen format that contains a graphics (or large text) window and no text window.

function

A command built into the BASIC language that performs a computation on a constant or variable; for example, the square root function, or the absolute value function.

graphics

1) A mode of operation that displays pictures instead of text, or

2) the pictures displayed in this mode.

graphics characters

See control graphics characters.

graphics mode

One of several different formats for producing pictures on the screen.

graphics window
In certain of the graphics modes, the part of the screen where the picture is displayed.

hard copy
Printed output, as opposed to temporary TV or monitor display.

hardware
The physical apparatus and electronics that make up a computer, or any device connected to the computer, such as a printer.

high resolution graphics
A graphics mode (8) that provides the greatest number of pixels; and therefore, the greatest detail (resolution) in the graphics picture.

hi-res
See high resolution graphics.

hires graphics
(Pronounced "hi-res") see high resolution graphics.

hue
1) Color, as red, yellow, or green, or
2) the second number in a SETCOLOR command.

immediate mode
See direct mode.

increment
(noun): the amount by which a variable is increased in a loop.
(verb): to increase the value of a variable in a loop.

incrementation
The process of incrementing a variable in a loop.

input
Data or information coming into the computer, either from the keyboard, disk drive, or cassette recorder.

initial value
The first, or starting, value assigned to a variable in a program or in a loop.

initialization
The process of assigning starting values to variables in a program or in a loop.

interactive program
A program in which the user (person using the program) must make responses to messages on the screen at various points in the program.

inverse video
Dark text on a light background, as opposed to the normal light text on a dark background. Activated by pressing the inverse video key.

keyword
A word that has meaning as an instruction or in a command, and thus must not be used as a variable name or at the beginning of a variable name.

kilobyte (abbreviated K)
1024 thousand bytes. "48K memory", for example, means 1024×48, or 49,152 bytes (characters) of memory.

line number
A number (integer) that identifies a particular program line in a deferred mode BASIC program.

logical line
A numbered statement in a program. It may consist of a maximum of 114 characters (3 physical lines).

loop
A sequence of statements in a program that is executed more than once.

lower case
Small letters, as opposed to large (upper case) letters.

luminance
1) the brightness or darkness of a color (hue), or
2) the third number in a SETCOLOR command.

matrix
A two-dimensional array. Matrices may be thought of as "grids" with rows and columns. The individual elements in a matrix may be referred to with double-subscripted variables.

memory
The part of a computer (usually RAM or ROM) that stores data or information.

memory location
A number that refers to a particular place in the computer memory.

microprocessor
See central processing unit.

mode
A style of operation, as a graphics mode (as opposed to a text mode), or immediate mode (as opposed to deferred mode).

nested loops
Two or more loops placed inside one another so that the innermost loop is completed before the next innermost, and so forth.

numeric variable
A label for a memory location which can store a number.

operator
See arithmetic operator.

output
Data or information coming from the computer, either to the screen, printer, disk drive, or cassette recorder.

paint pot
Our nickname for color register.

physical line
One line on the screen, 38 characters long.

pixel
Picture element. One point on the screen display. The size of a pixel depends on the graphics mode being used.

print zone
A built-in column arrangement for printing text on the screen or on the printer. There are 4 zones (columns) on the screen.

program
A sequence of instructions to the computer that describes a process. A program must be written in a language (BASIC, PILOT, PASCAL, etc.) that the computer can understand.

program recorder
A special cassette recorder used to store programs on cassette tape.

prompt
A printed message on the screen that indicates that the user must type a response.

radian
A unit for measuring arcs (or central angles) of a circle. An arc of one radian has a length equal to the radius of the circle, and the corresponding central angle has a measure of pi/180 (approximately 57.295) degrees.

reserved word
See keyword.

RUN mode
See execute mode.

scientific notation
A system of expressing numbers as a product of a power of 10 and a number between 0 and 10. For example, the number 32,767 is written in scientific notation as 3.2767×10^4.

screen editing
The process of changing text on the screen by means of the cursor control keys.

screen format
The size of the screen (rows and columns) for a particular graphics or text mode.

software
Programs and data, as opposed to hardware.

sound channel
1) One of four memory locations that is used to store sound information. Also referred to as "voice."
2) The first of the four numbers in a SOUND command.

special function keys
Keys that do not print characters by themselves, but that control the screen editing, such as the INSERT key, or the DELETE BACKSPACE key.

split screen
A screen format that contains a graphics (or large text) window and a smaller text window at the bottom of the screen.

statement
An instruction to the computer usually containing a line number (for deferred mode), a keyword, variables and/or constants, and the RETURN command.

storage
1) The computer memory, or
2) the space available on a disk or cassette for storing data.

string
A sequence of letters, numerals, and other characters. A string may be stored in a string variable.

string literal
A string that has a fixed value, as opposed to a string variable. For example, "Date of Birth" is a string literal.

string variable
A label for a memory location which can hold a string. For example, if DOB$="Date of Birth", then DOB$ is a string variable.

subroutine
A part of a program that may be executed at various points in the program, and that can be executed by giving a GOSUB command. The program lines for the subroutine need be written only once.

subscript
A positive integer attached to a subscripted variable to distinguish the various items in an array or matrix.

subscripted variable
A single variable name which may be used to refer to several different values by attaching a subscript to the variable name.

text mode
A screen format for displaying printed information (text), as opposed to a graphics mode. Three text modes (0,1,2) are available on the ATARI.

text window
Four lines at the bottom of the screen reserved for displaying text.

trace
(noun): a chart which shows the changes in the values of the variables and the output as each line of a program is executed.
(verb): to make such a chart.

transfer of control
See branch.

truncate
To cut off without rounding. For example, the number 2.71895 truncated to two decimal places would be 2.71.

upper case
Large (capital) letters, as opposed to small (lower case) letters.

variable
See numeric variable and string variable.

variable expression
A combination of variables, numbers, and operators (+, -, etc.) that can be evaluated to a single quantity. The quantity may be a string or a number.

voice
See sound channel.

unconditional branch
A branch in which there is only one option: control always passes to the same line.

zone
See print zone.

zone spacing
A printing option in which items are printed using the built-in column spacing, as opposed to compact spacing.

Index

$15.95

John M. Reisinger

A+ Programming
in Atari BASIC

Learn BASIC programming on the ATARI® 400™, 800™, and 600XL™ Home Computers with this excellent classroom-tested book. The author uses many fun and entertaining experiments and exercises in his step-by-step approach to learning the concepts of BASIC programming. The traditional business-related applications of BASIC have been deemphasized in favor of more motivating graphics and sound programs. Best of all, this book encourages you to program on the computer through plenty of sample programs, suggested program modifications, activities, and exercises. **A+ Programming in ATARI® BASIC** is the perfect learning tool in any setting—for self-study, or in schools, workshops, or computer camps.

Table of Contents

Operating the Computer • Printing • Graphics and Sound • Assignment (LET) Statements • The INPUT Statements • READ, DATA, and RESTORE Statements • Transfer of Control Statements • Anatomy of a Loop • FOR . . . NEXT Loops • Flowcharts • Writing Understandable Programs • More About Graphics • Numeric Functions • String Functions • Variables with One Subscript (Arrays) • Variables with Two Subscripts (Matrices) • Subroutines • Review

0-8359-0004-5

0   3

21898 00045